PC/GEOS

System Software Overview

DRAFT VERSION

10 November 1990

PC/GEOS Overview

PC/GEOS is a state-of-the-art graphical operating system that combines leading edge software technologies with numerous innovations in a tightly coded system software package. PC/GEOS runs well on 8088 machines with as little as 512 Kbytes of memory. On more powerful 80286 and 80386 machines, PC/GEOS takes advantage of extended memory and delivers nearly instantaneous performance.

One of the many technological breakthroughs in the PC/GEOS system is an object-oriented user interface technology that aides rapid application development while isolating applications from specific user interface designs. Currently three user interface libraries have been developed for PC/GEOS: Motif, Open Look, and CUA/Presentation Manager. A stylus-based user interface library is under development. Single application binaries designed for the PC/GEOS platform will run unmodified under each of these interfaces. Applications contain user interface objects that generically describe the required user input/output. At run-time these objects are dynamically modified into the specific visual objects appropriate for the chosen user interface.

Much of the compactness and flexibility of the PC/GEOS system is the result of GeoWorks' Object Assembly™ system. The Assembler for PC/GEOS is a superset of the Microsoft Assembly format with Pseudo Ops for supporting class definitions. The assembler is designed to work with the object messaging system built into the operating system kernel. This approach combines the structural benefits of object-oriented programming: data and code encapsulation, inheritance, reusability, etc. with the efficiency of having object methods (the procedures that act on messages sent to an object) coded in assembly language.

PC/GEOS supports true pre-emptive multitasking, including multiple threads of execution within a single process; a robust single imaging model complete with outline fonts, splines, polygons, etc. (comparable to PostScript and OS/2 GPI); dynamic memory management; and a windowing system that supports nested, overlapped, arbitrarily-shaped windows. While PC/GEOS has its own file system interface routines, these routines use MS/DOS for file access. This ensures compatibility with the wide range of disk devices used in the PC Compatible marketplace.

Components of the System

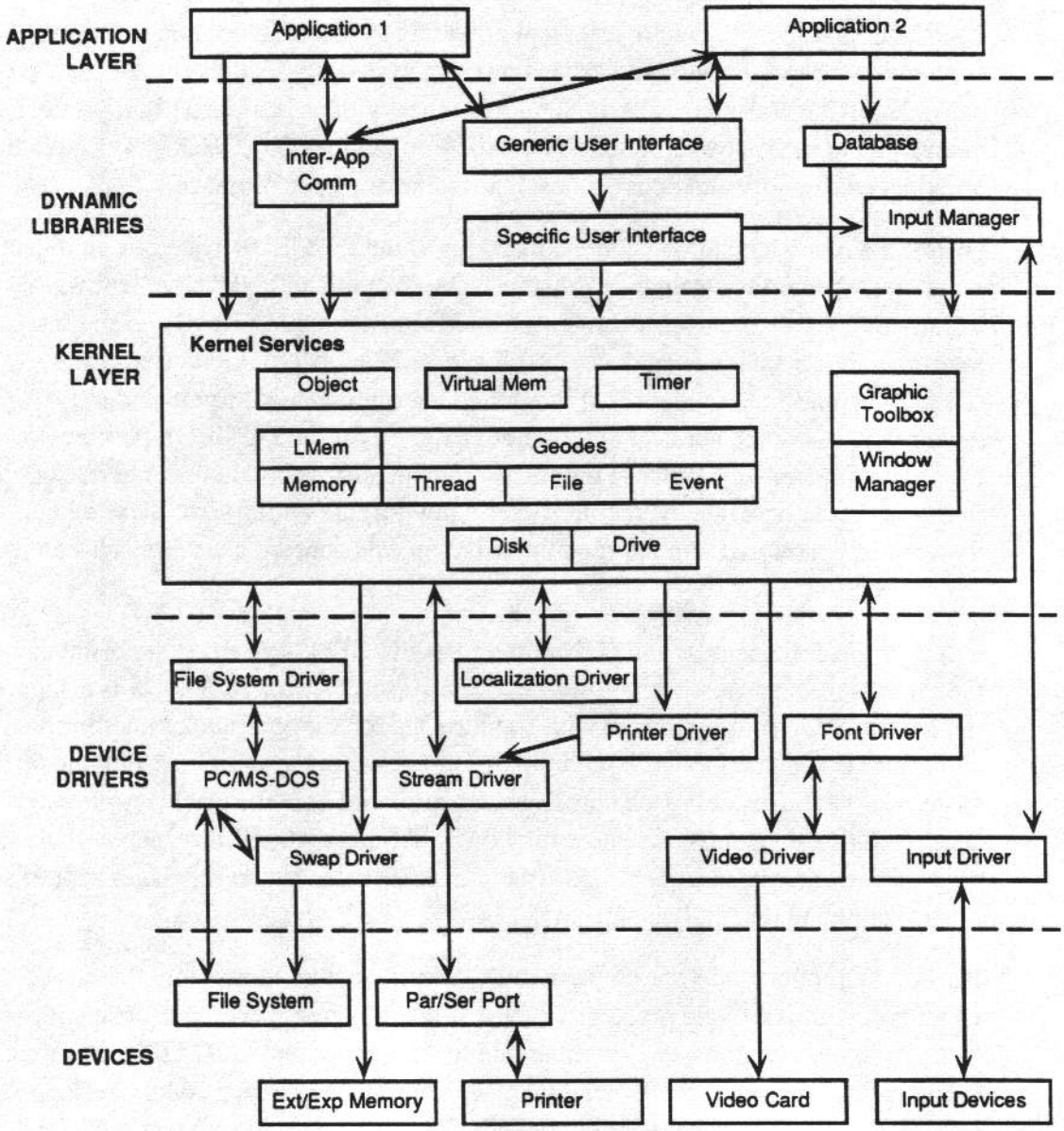


Figure 2. PC/GEOS System Architecture

There are several layers in the PC/GEOS system architecture. The top layer is the application layer. Dynamic libraries comprise the next layer. Libraries allow commonly used code to be shared amongst many applications. When an application is loaded, calls to library routines are automatically "linked" by the system. The kernel provides basic operating system services and communicates with the device driver layer. Device drivers function as an interface between the kernel and physical devices. For example, applications draw graphics and text to a resolution independent document coordinate

system. Device drivers are used to map these images to different screen display devices and printers.

The PC/GEOS kernel is the core of the operating system. It provides the following services: pre-emptive multi-threaded multitasking, dynamic memory management (both global and local heaps), process management, file system access, inter-process communication, object support (instantiation, messaging, inheritance, etc), and all of the imaging and graphics primatives. While there are over 500 globally accessible routines in the Kernel, most application developers only need to learn a small subset of these routines. The Kernel uses a 55K fixed block of memory and has 20K of library routines loaded as needed.

PC/GEOS libraries are modules of executable code which are dynamically loaded into memory when needed. A library allows commonly used code to be shared amongst several applications. Several standard libraries are provided with the operating system and are available to all developers. These libraries provide user interface components, a text-edit object, a file system dialog object, a print control object, database support, and other important functionality. Additionally, developers may create their own libraries.

A device driver provides applications, libraries, and kernel routines with a well-defined interface to a given type of device. Each driver contains code to control a specific device. Drivers are provided for output devices (video displays and printers), input devices (mice and keyboards), I/O ports (serial and parallel), and other device types. While most applications interact with drivers indirectly via kernel and library calls, applications may call a driver directly.

The PC/GEOS user interface library manages the user's interaction with an application. This library provides the capability to display windows, user controls, and visual output specified by the application.

All elements of PC/GEOS are efficiently coded in assembly language or Object Assembly™ and optimized for fast execution speed and economical memory space usage.

Object-Oriented Programming

PC/GEOS is an object-oriented operating system. While application developers are not forced to develop using object-oriented techniques, it is encouraged. This section is intended to give an overview of object-oriented terms and principles as a foundation for the following discussions. Those who are familiar with object-oriented technology may skip this section. However, a quick perusal might be valuable since different object systems often use different terminology.

There is nothing magical about object-oriented programming. Anything that can be done in an object-oriented system can be accomplished using standard procedural coding techniques and vice-a-versa. The principles of object-oriented programming simply provide a structure for making code more reusable and for isolating individual coding elements of large systems. This section will explain some of the basic terms and principles of object-oriented programming: data and code encapsulation, class definition, multiple instantiation, messaging, sub-classing, and inheritance.

An object is simply a grouping of data and routines that act on that data. For example, in programming a card game, a programmer working in an object-oriented system may choose to make each card an object. A card object might have routines to draw the card to the screen, to flip which side is up, or to change the card's position. The card object would have data to indicate the type of card (number and suit), what side of the card is currently up, card position, etc. The internal details of the card object data structures and the routines that manipulate those data structures are inaccessable to other objects in the system; this is called ***encapsulation***. Data and code encapsulation are an important part of object-oriented systems. To get a card to flip over, a ***message*** is sent to the card object. For example, the game rule object might send a message to a card object to have that card flip over. The message is simply a number that is associated with a routine in the card object. Internal to the object, the message is matched to a routine that manipulates the object's data to effect the change. The procedures that process messages sent to an object are called ***methods***.

In a standard card game, there are 52 cards and so in the above example there would be 52 card objects. It would not make sense for each card object to have its own set of routines to operate on the card data since these routines would all do the same thing. To solve this problem, while maintaining the desired data and code encapsulation, object-oriented systems implement what is called a ***class***. Objects with identical data structures and code (methods) are grouped into a class. Instead of each object containing its own copy of the routines that operate on that object's data, there is a pointer to the class definition where the routines or methods for the class are defined. This way, the code for a given ***class*** of objects is shared across each ***instance*** of that class.

Class Definition is central to object-oriented systems. Objects are actually created by calling an ***Object Instantiation*** routine with a pointer to the class definition for the desired object. In addition to the routines or methods for an object class, the class definition contains information on the structure and amount of data required to create an ***instance of that class***. Conceptually, a class definition for a card object would contain the following elements:

```
CARD OBJECT CLASS DEFINITION

CardObject
          ptr MetaClass      /* pointer to super-class */
                                        (explained later)

          CardMethodTable    /* message, pointer to routine */
             INITIALIZE_MSG, ptr Init_Method
             MOVE_MSG, ptr Move_Method
             FLIP_MSG, ptr Flip_Method
             DRAW_MSG, ptr Draw_Method

          CardObjectDataStructure
                             /* instance data definitions */
             int xPosition, yPosition
             int value
             enum[heart,diamond,club,spade] suit
             bool faceUp
             ptr cardImage
```

To create an instance of a card object, a pointer to the CardObject class definition is passed to the ObjInstantiate procedure. An instance of the CardObject class would conceptually contain:

```
INSTANCE OF CARD OBJECT

CardInstance
          ptr CardObject     /* pointer to class definition */

          xPosition = 52     /* instance data for this object  */
          yPosition = 96
          value = 13
          suit = heart
          faceUp = TRUE
          cardImage = KingOfHeartsImage
```

One of the most powerful innovations delivered by object-oriented systems is the ability to modify the behavior of an object class without having to create an entirely new class.

Returning to the deck of cards, suppose that after completing the game design, the programmer decided to add a feature where the face cards "winked" at the player. One way of accomplishing this might be to add the capability to the card object. The data structure for the wink picture and the routine to draw the winking effect could be added to the class definition. The disadvantage of this approach is that all of the non-face card object instances would have the data structures necessary to accomplish the winking effect even though they didn't use that data (remember, every instance of a class gets the same data structure). Since there are only 12 face cards in the deck, this would waste a lot of memory. Another approach might be to create a whole new card class just for face cards. The drawback here is that all of the basic routines to move the card and put its bitmap up would have to be duplicated in the face card object. Object oriented systems get around this problem using a technique called ***sub-classing***.

When a programmer wants to slightly modify the behavior of an object class, a new object class, called a ***sub-class,*** can be defined. For example, a new face card class can be defined as a sub-class of the card object. To support sub-classing, all object definitions contain a pointer to their ***parent*** or ***super-class.*** In the face card class definition, only the new data structures required to support the face animation and the new method required to support the wink effect are defined. When an instance of the face card class is created, however, storage is allocated for all of the data structures of the super-class as well as for the new data structures defined. This is conceptually illustrated below:

```
FACE CARD OBJECT CLASS DEFINITION

FaceCardObject
        ptr CardObject     /* pointer to super class */

        FaceCardMethodTable
           WINK_MSG, ptr WinkMethod

        FaceCardDataStructure
           ptr winkImage
```

Notice that the class definition for the face card class is very compact. It only contains a pointer to the super-class for the object and the new method and data structure necessay to create the desired effect. However, an instance of the face card class has the full data structure of the super-class as well as the data structure added by the face card class definition:

```
INSTANCE OF FACE CARD OBJECT

FaceCardInstance
        ptr FaceCardObject   /* pointer to class definition */

        xPosition = 52     /* instance data for superclass */
        yPosition = 96
        value = 13
        suit = heart
        faceUp = TRUE
        cardImage = KingOfHeartsImage
                          /* instance data for FaceCardClass */
        winkImage = KingWinkImage
```

When a message is sent to a face card instance to move the card, the method table for the face card class is searched for a MOVE_MSG. Since the only message defined for the face card is WINK_MSG, the system follows the super-class pointer to the parent class and looks for the message in the method table there. This organization of classes and sub-classes in referred to as a ***Class Hierarchy.*** The concept of a sub-classed object having the data structures of its parent class and following the super-class link to process messages not found in the class definition is called ***Inheritance.*** A subclassed object is said to inherit the properties of its parent or super-class.

Object Assembly

PC/GEOS is a complete object-oriented operating environment. Most of the system software is implemented in Object Assembly™, a GeoWorks innovation. Object Assembly combines the structure, data types, type checking, and class definition of a high level object-oriented language with the compactness and performance of assembly code. While programmers will not be required to program in Object Assembly or use object-oriented programming at all, GeoWorks encourages programmers who are familiar with 80x86 assembly to seriously consider Object Assembly as a development environment. For traditional developers a "C++" and "C" development environment will be supported.

ObjAsm is the assembler for the PC/GEOS environment. It is a superset of the Microsoft Macro Assembler (MASM), i.e. it will assemble any code written for MASM but it contains many advanced features. It provides all of the data types available in "C" including: signed and unsigned integers, enumerated types, structures (which may be arbitrarily nested), unions, bitfields, pointers, arrays, and named types that are arbitrary combinations of any of these. Type checking is also supported. ObjAsm provides extensive support for the object model built into the PC/GEOS kernel. This includes the definition of classes, their instance data, messages, and the methods that respond to messages. GeoWorks considers Object Assembly a "medium" level language. It combines the structure and type checking of a high level object-oriented programming language like C++, with the efficiency of having methods implemented in assembly language.

To better understand Object Assembly, let's return to the card object used in the object-oriented programming example of the preceding section. The class definition in GeoWorks Object Assembly would look like this:

```
SuitTypes    etype    byte     ;an enumerated type to indicate suit.
ST_HEART     enum     SuitTypes
ST_DIAMOND   enum     SuitTypes
ST_CLUB      enum     SuitTypes
ST_SPADE     enum     SuitTypes

CardObjectState  record     ;this byte-length record contains state
                            ;information for a card object
          :1                ;unused (bit-length field)
          COS_FACE_UP:1     ;TRUE means face-up,
                            ;FALSE means face-down
          COS_SUIT:2=SuitTypes  ;card suit,
                                ;see enum definition above.
          COS_VALUE:4       ;card value, from 1 (ace) to 13 (king).
CardObjectState ends

CardObjectClass   class   MetaClass ;beginning of class definition

;messages:

INITIALIZE_MSG              method
MOVE_MSG                    method
FLIP_MSG                    method
DRAW_MSG                    method

;instance data definition:

    CI_xPosition    word              ;X position
    CI_yPosition    word              ;Y position
    CI_state        CardObjectState   ;state: see record
                                      ;definition above
    CI_cardImage    fptr              ;pointer to bitmap in
                                      ;same resource
CardObjectClass endc          ;end of class definition
```

To create an instance of this class, the programmer simply calls a routine in the PC/GEOS kernel with a pointer to the class definition and a pointer to the memory block into which the object will be instantiated.

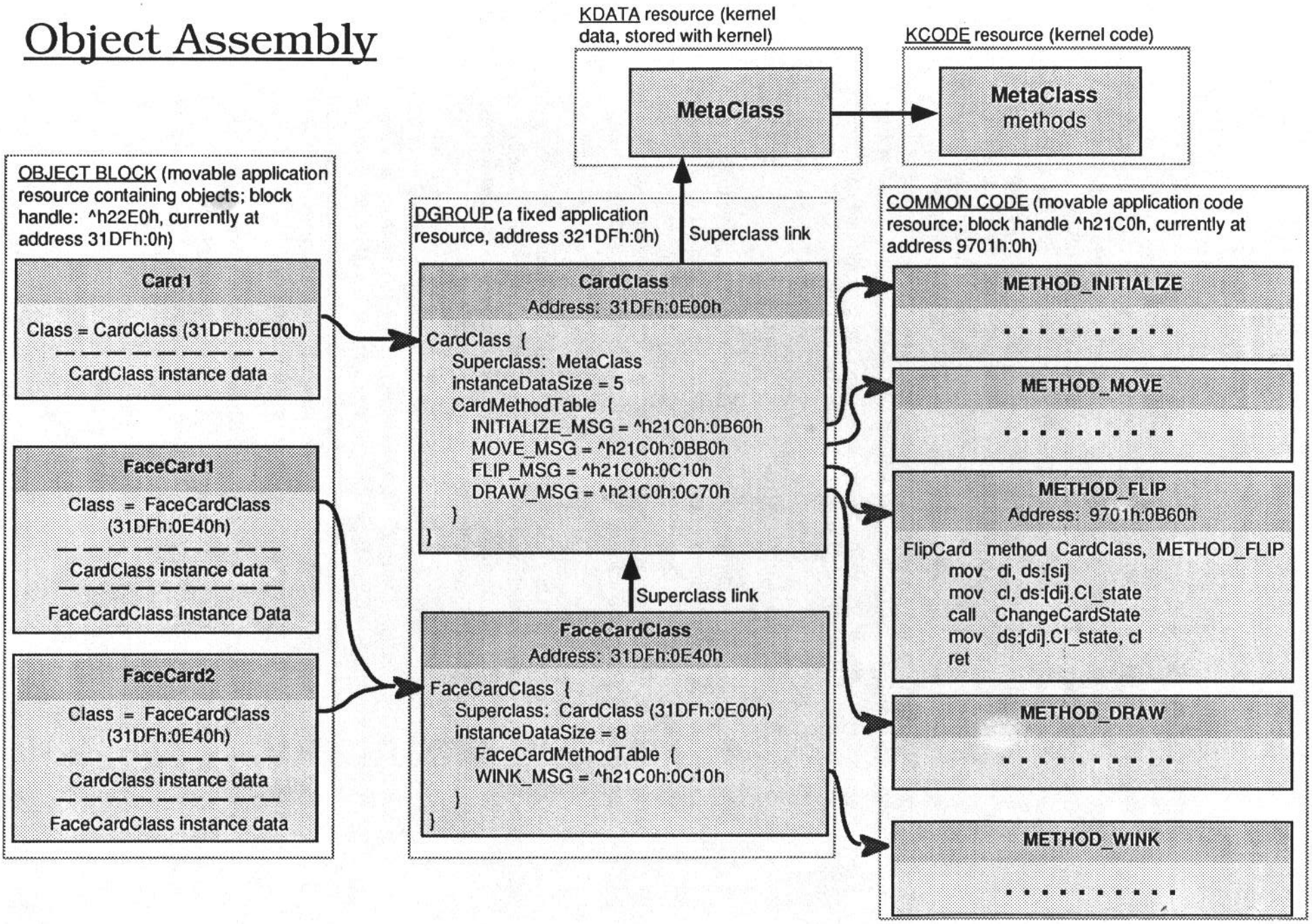
```
;create an instance of CardObjectClass,
;placing it into the object block
;whose handle is in BX.

mov     es, segment CardObjectClass
                ;set es:di = pointer to CardObjectClass
mov     di, offset CardObjectClass
call    ObjInstantiate
                ;returns ^lbx:si = handle of new object.
```

The face card sub-class definition would appear as follows:

```
FaceCardObjectClass   class   MetaClass
                          ;beginning of class definition

;messages:

WINK_MSG                  method

;instance data definition:

    CI_winkImage   fptr   ;pointer to bitmap in the same resource
FaceCardObjectClass endc
```

The diagram on the following page shows how the playing card object example would be implemented under PC/GEOS. This diagram shows the relationship between object instances, their class definition, and the inheritance relationship between a class and its super class in the PC/GEOS system

Object Assembly



Applications

Because PC/GEOS offers so many libraries and routines to handle everything from the user interface to graphics to file management, applications developed for this system are surprisingly compact. PC/GEOS frees developers from most of the busy work associated with graphical environments. The object-oriented user interface technology eliminates the burden of writing code to support common user interface elements such as menus and dialog boxes. Developers can graphically construct the user interface for their applications using a graphical builder. The resulting user interface object resource file is simply linked into the application. As a result, PC/GEOS applications are typically several times smaller than equivalent Windows or Macintosh™ applications. GeoWrite, the wordprocessor included in the GeoWorks Ensemble package, is a 57 Kbytes file including overhead for the application icon and code relocation information, the total executable code is under 50K.

The Structure of an Application

PC/GEOS applications are primarily object-oriented, but they don't have to be. You may decide to take full advantage of the object-oriented nature of PC/GEOS, creating classes of objects, such as a deck of cards, which model the task at hand. Or as in Windows, a PC/GEOS application may be written as a single monolithic "case" statement, handling specific messages sent by the user interface and other system components.

Whichever scheme is chosen, the most important aspect of the application to the user is its interface. The application's user interface is composed of objects which are created and managed by the user interface library. These objects are instances of various user interface classes and implement windows, menus, buttons, and so on.

In addition to these user interface (UI) objects, an application also has a "ProcessClass" object. The ProcessClass object ties the application together. It contains methods which respond to messages from the UI and the operating system. The UI objects send messages to the ProcessClass object to inform it of such events as a mouse-click or the selection of a menu item. The operating system sends it messages to notify it of system-related events, such as when the application is first executed.

Such an arrangement results in the separation of the user interface from the internal functionality of an application. This separation lends itself naturally to the execution of the ProcessClass object and UI objects in different threads (see the "Threads" section for more detail). The ProcessClass object is run by the application's thread and the user interface objects are run by the user interface thread. Therefore, the user can still interact with the application even when it is busy recalculating values or preparing to print.

All objects are instances of pre-defined classes supplied by the libraries available to an application. The GenTrigger class is an example of a UI class. The GenTrigger class contains all of the state data and functionality required for a simple pushbutton in an application. One of its state variables is **GENS_ENABLED**, which determines whether the GenTrigger object is enabled or disabled — whether the user can interact with it or not. The functionality associated with the GenTrigger class is simple: when the user activates the button by clicking on it with the mouse, the button sends a specified method to another object, typically the application's ProcessClass object.

As discussed in the previous section, an application may define subclasses of classes available in libraries. Most common is the subclassing of user interface components to modify their behavior to the application's needs. For instance, a solitaire game may subclass a UI component (VisClass) to simulate each individual playing card.

ProcessClass Object

The ProcessClass object stores information about the state of the application, contains most of what can be considered the core code of the application, and to some degree coordinates the interaction between other (primarily UI) objects. The ProcessClass object handles messages sent from UI objects as well as other components of the system. As these messages are received, the internal state of the ProcessClass object can be examined and changed. For example, the ProcessClass object might respond to a message from a button which has been triggered by creating a new window, drawing some graphics objects, or saving a file to disk.

The ProcessClass is not the only object an application may instantiate. An application can consist of many objects, all run by the application's thread of execution, along with the ProcessClass object.

Additionally, an application may have several independent threads of execution, to handle timing-critical tasks (such as monitoring the serial port), and background tasks, such as recalculating a spreadsheet.

Output and Action Descriptors

UI objects send messages to each other and to the ProcessClass object using output and action descriptors. An output descriptor is a two word field describing the destination for a message. The action descriptor is composed of a word describing the action to take and an object descriptor. These descriptors are found in the definition of UI objects as well as in code segments.

Resources

Resources are bundles of application routines or data which are loaded in and out of memory independently as they are needed. How an application defines resources explicitly affects the execution of the application. Every routine and every object resides in some resource. You should choose the grouping based on how you envision the code being executed, with an eye toward maximizing efficiency. For example, you might group your code by functionality: common code used by all parts of the application, code which initializes variables and is only run once, code associated with the UI, and obscure code which is rarely used. In this case, the common and UI resources would probably remain in memory, the initializing resource would be loaded once and discarded, and the obscure resource would be swapped in only when needed.

Source Files

A PC/GEOS application consists of four basic types of source files. These include the geode description file, the definitions file, the user interface file, and the code file.

Geode description file (.gp): This file is used by the linker. It specifies the type of "geode," or executable, which should be created — application, driver, or library. The file also includes details such as the 32 character long name of the PC/GEOS file, the libraries required, the name of the topmost user interface object, resource definitions, and so on.

Definitions file (.def): This file contains constants, structures, records, and classes used throughout your code. It is not required, but is suggested for larger applications. If you have multiple **.asm** files (described below), this file relieves you of having to redefine all of your declarations in each individual **.asm** file.

User interface description file (.ui): This file contains a hierarchical description of the application's user interface in generic terms. This description is written in a high-level language called UIC, which is compiled and then linked into the application.

Code files (.asm): This file contains code to handle methods sent to the ProcessClass object. An **.asm** file may also include definitions not placed in a **.def** file. A complex application may have several **.asm** files, perhaps one for each additional class of objects which is managed by the application. Multiple **.asm** files are linked during the compilation of the application.

Source code in an **.asm** file may be divided into sections, called modules, based on logical divisions in the methodology or functionality of the application. Separate modules are linked together to form the application. Modules simply clarify the organization of the program; they do not affect the execution of the program.

A large application may be composed of many **.asm**, **.def**, and **.ui** files.

User Interface

A user interface (UI) is a set of rules and conventions by which a computer system communicates with the person operating it. Initially, operating systems (such as UNIX or MS-DOS) featured text-based command line interfaces. Users were expected to use and remember complicated, forgettable commands such as "copy c:\utilities\disk\go.bat b:." Different applications all had different user interfaces — to print the current document, a user might have to press the function key F7 in a word processor and the keys Ctrl-Alt-P in a database program. Computers were difficult to learn, difficult to use, and, worst of all, inconsistent. In the pursuit of the often-coined property known as "user friendliness," much work was done in terms of improving user interfaces. Just as the personal computer market as a whole is changing rapidly and drastically, so too are user interface standards.

Through the years, operating systems have evolved from those complex command line interfaces to graphical, windowing environments such as the Apple Macintosh and Microsoft Windows. These new graphical user interfaces (GUIs) feature menus, buttons, and windows, accessed by a mouse. The graphical and intuitive nature of these interfaces solves many of the problems inherent in earlier operating systems. Graphical system software typically provides a large toolkit of user interface gadgets, such as windows, buttons, and menus. Applications make use of these UI items to implement their interaction with the user. In order to prevent the proliferation of inconsistent applications, companies develop rules and conventions for using the UI gadgets. Documents known as style guides are provided in an attempt to instruct application designers in the appropriate usage of the user interface gadgets offered by a system. Some examples of such user interface standards are OSF/Motif, OpenLook, CUA, NewWave, and Macintosh. GeoWorks refers to each of these standards as a specific user interface.

However, even applications developed for a user friendly environment like Windows or Macintosh can be difficult to use. As applications have become more and more powerful, they have also become more and more difficult to use. There are so many fascinating and complex things users can do with these new programs that it's simply impossible to create a user interface that is always easy to use. A new concept in the GUI community attempts to come to terms with this problem. It is the scalable graphic user interface. Such a GUI allows the same applications to be accessed at various levels of functionality. These levels range from an appliance mode, where users are only required to push a few buttons, to a full-fledged professional graphic user interface like Motif. Users, as their skills and needs grow, may simply switch interface levels to access more powerful features. So if users only want to quickly type a letter or envelope, they do not have to wade through a program designed to produce newsletters involving multiple columns of text running from page to page and graphics placed

randomly throughout the document. They can merely run the word processor in appliance mode and painlessly type a simple letter without having to set a lot of options and to pick their way through a hundred extra features.

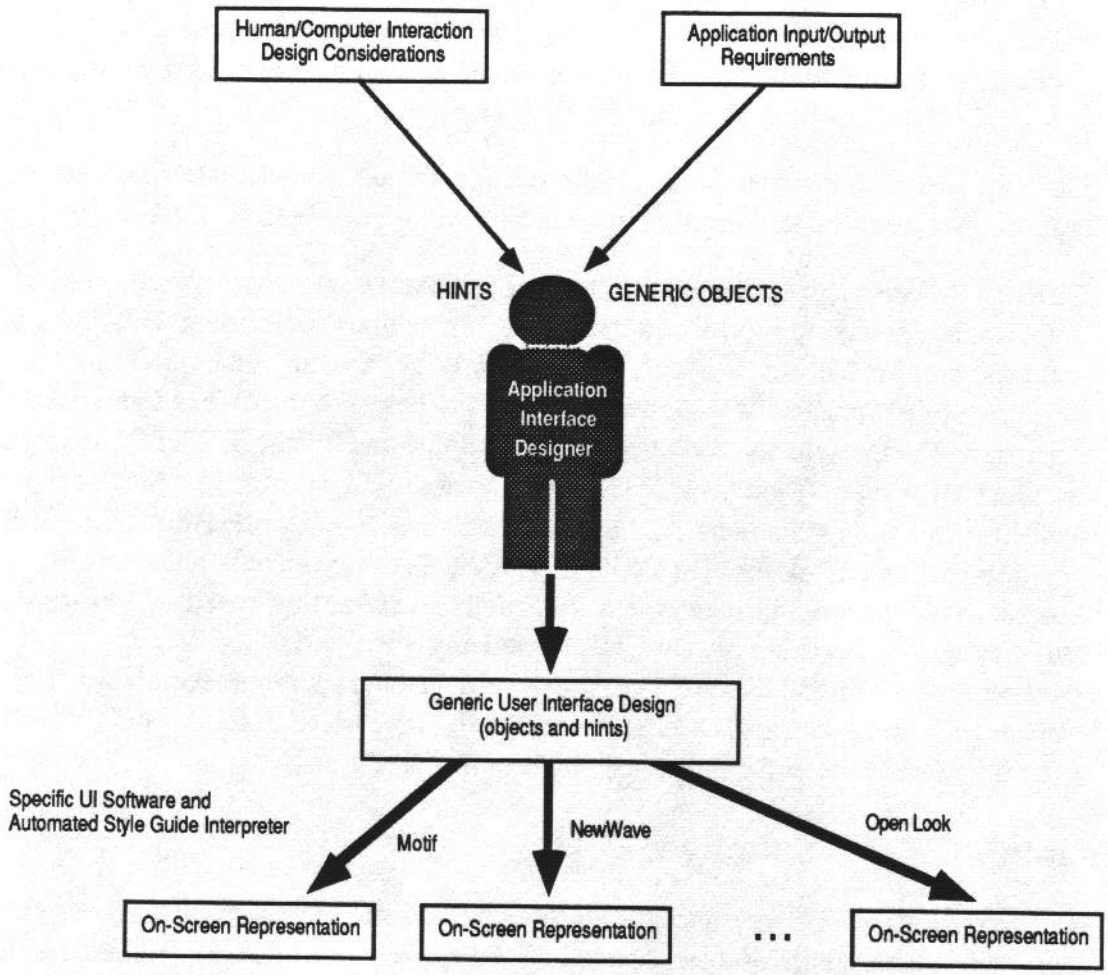
From the user's standpoint, PC/GEOS provides two important user interface features: a scalable user interface (as described above) and user-selectable look and feel.

Different look and feel options permit the user to choose between several commercial graphical user interface specifications, running applications conforming with the visual and behavioral guidelines of Motif, CUA, Open Look™, or any other GUI. So users may use whichever specific user interface they like best. For instance, a user who runs Microsoft Windows on his '386 at work can go home and instruct PC/GEOS to employ the CUA style guide. Then his PC/GEOS applications will have the same look and feel as their Windows applications. And his wife, who uses a Sun workstation at the office, could select Open Look for her PC/GEOS session. The very same applications her husband executes would have the same look and feel as her Sun-based applications. But they are not limited to current GUIs. Given any user interface specification developed in the future, GeoWorks can provide a specific user interface library. The same application executables users previously ran under Motif or CUA would then perform properly under the future specific user interface.

Developer's Perspective

User interfaces have always progressed toward more and more user friendly designs, but often at the expense of those developing the applications. PC/GEOS addresses this issue, making user interfaces not only user friendly but "developer-friendly."

GeoWorks has redefined how application user interfaces are developed. The designer does not attempt to define the final, gadget-level interface to the application. Instead, he selects objects from a generic UI object library based solely on the input/output requirements of the application, and groups them according to function within the application. Subjective design considerations associated with those requirements, which would ordinarily be weighed by the designer in order to pick specific gadgets, are instead stored digitally (as hints) along with the generic UI objects. The designer's job is done, as everything short of the style guide has been considered and stored as part of the UI specification for the application. This data is later interpreted in software by any one of a number of UI interpreters, which map the selected generic UI objects and hints into an interface implementation which meets the specific UI's style guide requirements. The final interface for the application is then presented on-screen.



GEOS Application Design Process

This approach allows the same application executable to come up with the look and feel of any number of specific user interfaces, meeting the style guide requirements and recommendations for each. The more information about the application's interface requirements and subjective considerations that can be stored in generic object and hint format, the better the interface that can be created for the application when running under UI interpreters for different or new specific user interfaces. Since the generic model essentially decouples the application from its user interface, the application is completely independent of changes in specific user interfaces. The application's user interface is specified solely in terms of common semantic properties rather than specifics of particular UI gadgets, so the application's user interface can be properly constructed and presented under *new* and *different* specific user interfaces. New UI interpreters for new style guides can be written *after* the creation of an application executable, and the application's user interface will be presented in accordance with the new style guide. What this means is that new, improved user interfaces could add novel and wonderful capabilities far beyond that imagined by the original application designer, simply because functional as well as subjective information about the

application's UI needs are stored with the application. Similarly, specific user interfaces intended for users with varying levels of proficiency may be defined, so the very same application executable can also be presented appropriately to both novice and advanced users.

Once the application's user interface is described in generic terms, PC/GEOS maps each generic UI object to one or more specific UI objects, depending on which specific user interface the user has chosen. For example, an application's UI file might specify that a list of options be presented to the user. Depending on the attributes and "hints" of the generic object, this might be implemented as a menu button in OpenLook or as a scrolling list in OSF/Motif™. The conversion from generic to specific user interface is transparent to the application. PC/GEOS can accommodate any number of specific user interface libraries, though OSF/Motif is the only one presently available.

Defining an Application's User Interface

An application defines its user interface using generic UI classes.

Generic User Interface Classes

Generic UI classes are abstract types of user interface components. By thoroughly researching and analyzing existing and proposed GUIs, GeoWorks identified the major kinds of user interface components that were common. Abstracting these components — reducing them to their functional essence — resulted in ten generic UI classes. For example, all specific UIs need a method of initiating an action — hence the generic trigger class. The UI library file **generic.uih** defines these classes. You must include this file in a program's **.ui** file. The names of generic UI classes are all prefaced by "Gen."

GenApplication

The GenApplication class represents the highest level of management for an application's UI components. It generally manages the various top-level windows of the application and does not have a visual representation. Every application must have an instance of this class. It is the top-level node of a tree structure containing all of the application's UI objects. PC/GEOS places this object in its own resource, so that it may be independently moved in and out of memory.

GenPrimary

The GenPrimary class provides the primary means of accessing the application. In most specific user interfaces, this is the main window for an application. A GenPrimary object groups and manages all of the controls and output areas that appear when the application is launched.

GenTrigger

The GenTrigger class allows a user to trigger a particular action. It may be implemented as a pushbutton, menu choice, or even voice input which initiates a certain action when triggered by the user. The GenTrigger description in the application **.ui** file defines the message to send when activated, and the object to send the message to.

GenSummons

The GenSummons class is used in situations where a temporary dialog with the user is needed. In most specific user interfaces, the generic object becomes a dialog box. By default, the dialog box is "application-modal," meaning that the user must respond to it before proceeding with the application. Depending upon the object's attributes, the specific UI may create a button which the user can click to open the dialog box.

GenInteraction

This class serves as a generic grouping object. Depending upon the attributes specified for an instance of this class, a GenInteraction object can affect the organization and/or visual grouping of other generic objects. This object comes in three basic flavors:

As a group of controls: By default, a GenInteraction object is simply a visual box in which other generic objects may be placed. Its contents may be placed horizontally or vertically. A title and frame outline may be drawn if desired.

As a non-modal dialog box: A GenInteraction object with the **independently-Displayable** attribute set is implemented as a non-modal dialog box. Many applications use this object to contain controls that the user may want to keep on the screen, such as "Line Color."

As a menu or sub-menu: When the hint **HINT_MENUABLE** is specified for a GenInteraction object, it means that its contents should be readily accessible, though not so as to take up space on the GenPrimary window. This is typically implemented as a menu or sub-menu. The generic children of this object become menu items and are placed in the menu.

GenRange

The GenRange class allows users to interactively set a value within a discrete range of values. This object might be implemented in many ways by the specific UI: a slider, a spin gadget, a dial, and so on. The range values are signed, between –32767 and 32767. Increments are also between –32767 and 32767. Alternately, the GenRange object has a "distance" mode, where its value is assumed to be a distance and can be set in several units, such as inches or points.

GenList

A GenList object is used when the user must select a number of items from a list of many. Some of the gadgets available in different specific UIs which may be used to accomplish this are scrolling lists of items (of which one or more is highlighted), radio buttons (of which one may be selected), menus of items (in which the last one selected is checked), and so on. The functionality incorporated in these gadgets can be further abstracted as follows:

- exclusive — one and only one option may be "on" at a time. For example, from a pair of radio buttons specifying text size (small font, big font), the user should choose one and only one option.
- exclusiveNone — one option or none may be "on" at a time. For example, in a group of radio buttons controlling baud rate (300, 1200, 2400, etc.), the user may choose either one option or none.
- nonExclusive — each option may be turned on or off independently of the others, such as check boxes.
- nonExclusiveWithOverride — each option is independent of the others, except for one (the first) which is exclusive. For instance, in a list containing "plain, bold, italic, underline" the user may select bold, italic, and underline independently. However, selecting "plain" forces the other three options off.

The list of options may be displayed in a dynamic or static list. The file listing is an example of a dynamic, scrolling list used when the option list is subject to change and when there is too much information to display on the screen at once. Radio buttons controlling the print quality (high, medium, low) are static; they will not change.

GenView

This class provides the application with an area of the screen on which a document may be shown. The application simply provides a document (see below); the GenView manages everything else: creating and managing the scroll bars which permit the user to scroll the view over portions of the document, scaling the document to provide "close-up" or "far-away" views, and updating the screen as other application windows are moved off of the view area.

The document, the actual contents of the view, may be created simply by calling the kernel's graphics toolkit routines to draw the lines, text, and bitmaps that make up the page. As the view scrolls, the user interface sends a **METHOD_EXPOSED** to the application so that the application may redraw the portion of the document which has been uncovered by the scroll movement.

A document may also be described by a tree structure of "visual" objects. GeoDraw is a convenient example of this usage: each graphic element is a visual object which may be moved and resized. As the view requests portions of the page to be redrawn, only the visual objects in that area are forced to redraw. As an optimization, the application designer may decide whether these visual objects are run by the UI thread or the application's thread.

Displays

The GenDisplay class is used to manage a GenView object. It provides the various accoutrements required by the chosen specific user interface (window controls such as the maximize and minimize buttons, etc.). The GenDisplayControl class manages several GenDisplay objects. In a "professional" user interface, such as Motif, the GenDisplayControl creates a "Windows" menu, containing items such as "overlapping" and "full-sized," allowing the user to manage multiple document windows. For example, GeoWrite uses a GenDisplay object for each document opened and manages the document windows with a GenDisplayControl object.

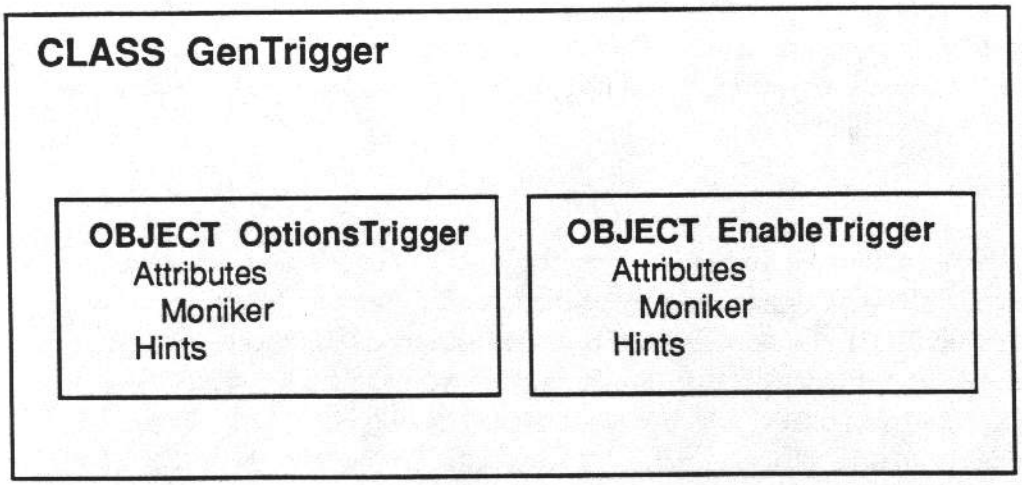
Text Objects

Whether an application requires a one-line text field or a multiple-page document, the GenTextEdit class is used. This UI object handles differently formatted text, keyboard navigation, cut and paste, and any other functionality required for a field of text.

The GenTextDisplay class is utilized when an output-only text object is required. For example, an adventure game might require text to be displayed which the user should not modify.

Generic User Interface Objects

Generic UI objects are instances — specific incarnations — of generic UI classes. So when an application needs a particular UI component (a button, for instance) it chooses the appropriate generic class (**GenTrigger**) and asks GEOS to create an instance of that class. The application can then use the resulting generic UI object as part of its user interface. Each individual UI object has its own instance data whose scope is determined by the UI class. There are two kinds of instance data: attributes and hints.



Generic User Interface Objects

User Interface Components

When an application needs a particular UI component (a button, for instance) it defines a generic UI object that represents the functionality inherent in the type of component desired. GEOS provides different types of generic UI objects which determine the general category of functionality wanted. Special properties of that object are set to convey more detailed as well as vague information about the human/computer interaction design considerations and application input/output requirements.

Basically, these generic UI objects are data structures with two different types of instance data — attributes and hints.

Attributes

Attributes define the behavior and/or appearance of a UI object in a very specific manner: an attribute is either on or off, and there is a definite set of attributes associated with every UI object class. When an application sets an attribute, it can be sure that the specific UI component that GEOS selects exhibits the desired behavior.

For example, setting the modal attribute for a dialog box ensures that the user must respond to it before continuing. Setting the disabled attribute for a trigger dims the trigger's label (called a moniker) and doesn't allow the user to select it.

Monikers

A moniker is a special attribute every UI object has. Each UI object may be given a *moniker*, or visual representation, though a moniker does need to be defined for every object. It could be the name of a button or the icon to be displayed when a window is minimized. A UI object is not restricted to a single moniker; a list of monikers may be defined. Depending on the situation and context, GEOS uses one of the monikers. For

example, an application may define different icons for CGA, EGA, and VGA monitors to optimize the its appearance. GEOS displays the proper one for a particular user's set up. Some UI objects may have several textual and pictorial monikers. GEOS chooses the appropriate moniker.

Hints

Hints provide additional information about the UI object in question. An application's needs are not always absolute and may be interpreted differently (even ignored) by different specific UIs. Some visual and behavior aspects of UI objects should not be implemented as attributes because of this. In other words, there are some UI components or functionality which is not universal to all specific UIs. Those capabilities cannot be attributes, since not all specific UIs support them. Therefore, they become hints. When the developer assigns hints to a particular UI object, he cannot be certain that the hint will be implemented by any one specific UI.

There are two types of hints: command and declarative.

Command Hints

Command hints are direct requests for a specific implementation of a UI component. A developer would choose to use a command hint when he had a specific UI component style in mind. For example, an application may explicitly ask for a scrolling list (**HINT_SCROLL_LIST**) or check boxes (**HINT_CHECKBOXES**). Not all specific UIs offer the capability to follow command hints. For instance, some specific UIs allow the user to use the keyboard to navigate menus and dialog boxes. To support this, certain UI objects would contain several **HINT_NAVIGATION_ID** and **HINT_NAVIGATION_NEXT_ID** hints. Motif would make use of this. OpenLook would ignore it because the style guide doesn't allow such navigation. GEOS fulfills a particular command hint in any specific user interface that supports it.

Declarative Hints

Declarative hints are more vague; without referring specifically to a particular implementation, they give an indication of the functionality of the UI object in question. For example, a generic UI object containing a list of possible actions may have a **HINT_MENUABLE**, indicating that the developer envisions the list being presented in a menu. However, perhaps a specific UI designed for novice users states that a menu is too complex. Then GEOS implements the list of actions as a simple series of large, plainly visible buttons. Or, similarly, an option in that menu may have hints stating that it is advanced, infrequently used, and potentially dangerous. Then a novice specific UI would remove the trigger altogether.

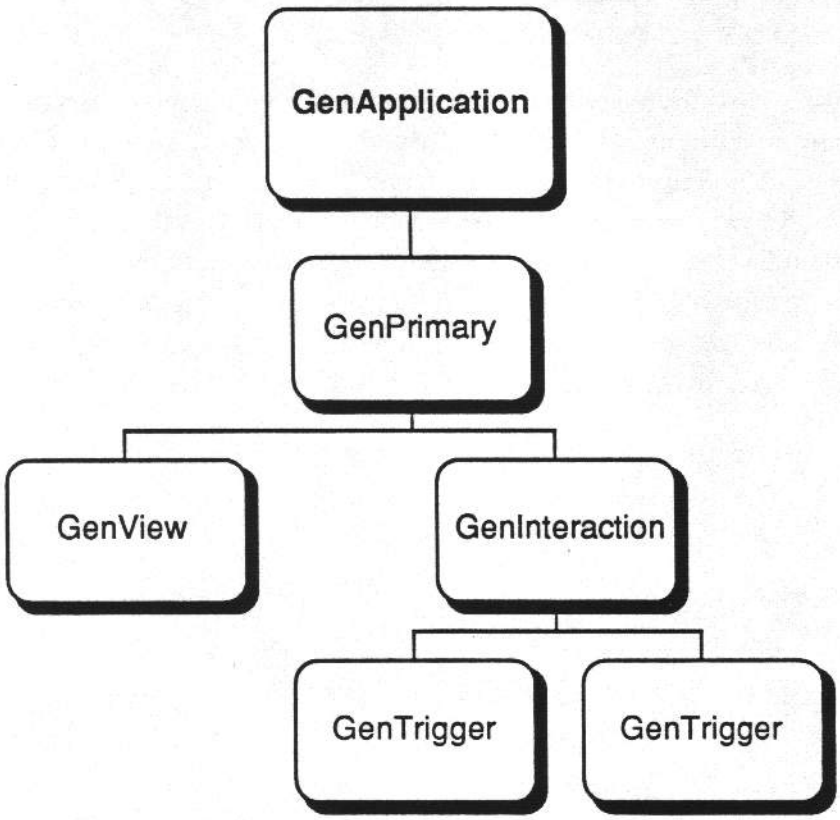
Once again, declarative hints may or may not be implemented by a particular specific UI. For instance, CUA does not allow submenus in the menu bar. A **GenInteraction**

object with the hint **HINT_MENUABLE** that is inside of another **GenInteraction** object with the hint **HINT_MENUABLE** would be implemented as a submenu in OpenLook or Motif. However, in CUA, it would be added to the menu and set apart by separators, since submenus are illegal according to the style guide.

The advantage of making the distinction between attributes and hints surfaces in memory usage and differences between specific user interfaces. Since hints are inherently dynamic — objects may have any number of hints — it would be wasteful to commit an attribute (and thus memory) to the behavior represented by the hint. By contrast, object attributes are necessary parameters for a given object class, so they are present in every object. Also, hints are not necessarily implemented by a particular specific UI, whereas attributes always affect the object. For example, **HINT_CHECKBOXES** creates special check boxes under Open Look but has no effect under Motif or CUA.

Using Generic UI Objects

The generic UI objects an application chooses to represent the UI components it needs are arranged into a tree. This tree is a hierarchy of UI objects, to convey the relative importance and interdependencies of each object. This provides an indication of which components ought to be in plain view and which can be hidden one or more layers deep. Below is an example of such a generic UI tree.



Sample Generic User Interface Tree

Managing UI Components

A traditional user interface management system, such as that provided by the Apple Macintosh, requires the application to control much of the interaction with the user. The application takes advantage of a library of gadgets (icons, scroll bars, buttons, etc.) but is required to realize these gadgets and control the user's interaction with them. When the user clicks a scroll bar, for example, the OS dispatches a click event to the application. The application must then decide how to respond to the click. PC/GEOS, on the other hand, manages user interaction for the application. For instance, an application with a scrolling view need not monitor the user's actions. When the user moves a scroll bar, PC/GEOS calculates what portion of the view has been exposed and sends the appropriate notification to the application.

Geometry Manager

Perhaps one of the more trying aspects of programming a graphically-oriented application is placing all the user interface components on the screen. At run-time, the Geometry Manager automatically organizes, formats, and displays the generic UI objects specified for an application. By defining each object's attributes and hints, you may fine tune the appearance of your application. You need not manually place each button, size every dialog box, and draw each frame and title.

The Geometry Manager provides many benefits. It eases the programmer's burden, automatically handling a very complex and time-consuming task. It adds another layer of abstraction to the PC/GEOS user interface, further shielding the application from the specific UI. With a global perspective, the geometry manager makes localization of applications to a certain foreign language much easier. The translation of text in the user interface to another language would traditionally require that many items be resized and moved. In PC/GEOS, however, the Geometry Manager immediately accommodates any such text changes, even resizing application windows if necessary.

Graphics System and Printing

The PC/GEOS graphics system provides a wide range of both vector and raster based primitives for drawing to the screen and for printed output. A single imaging model is used to generate images for both the screen and most output devices, resulting in as close a match as possible between the screen image and the printed page.

The drawing primitives available to application developers provide the same mechanisms as are available under other graphics packages such as Adobe's PostScript™ or OS/2's GDI™. The PC/GEOS command set has been expanded, however, to make simple images easier to construct. The following sections provide more detail about the specific functions available.

Applications create images using a device independent coordinate system. The graphics system maps those coordinates to the coordinates of a particular device. Similarly, the user may take advantage of the highest resolutions available on the print device, or use a lower resolution for quick output.

Coordinate System

The PC/GEOS graphics system is based on a 72 unit per inch (typographical point) coordinate space. This device independent system allows an application to specify an image to be an inch square, for example, and not concern itself with the particular hardware owned by the user. All of the output primitives expect units supplied in this coordinate base, and the graphics system scales the image to the correct units for the particular device being used.

In addition to working in a device independent coordinate space, the application writer has a number of functions available to manipulate the transformation from document coordinates (1/72nds) to the coordinate basis for the device. There are routines to apply additional scale factors (independent scaling in x and y), rotation by any angle (units supplied as fractional degrees), and support for other effects such as skewing.

Windowing System

The PC/GEOS windowing system, also part of the kernel, is closely tied to the operation of the graphic system. Its main function is to maintain information about the size, shape and ordering of the windows on the screen. Each application allocates one or more windows for its own use. The application calls graphics functions to draw into the window.

The application requires no knowledge of the size, shape, or position of the window. The graphics system transforms the drawing operations so that they appear in the

correct position on the screen. In addition to properly positioning the output, the graphics system does the necessary clipping to ensure that no graphics operations draw outside the area allocated for that application.

Some other features of the windowing system include the ability to open more than one window per application to display additional documents, and the easy addition of scroll bars via the user interface system to control what portion of the document is visible in the window. Each of these features requires little or no additional programming on the part of the application developer to support.

Graphics Primitives

There are four basic classes of output primitives available under PC/GEOS: text, lines, areas, and raster. Each class has a set of properties which controls the appearance of the primitive when it is rendered. There are a few properties that are common to all output primitives. These include:

- color, specified using 24-bit RGB values or 8-bit color indexes
- draw masks, applied to the image so that only a percentage of the image is drawn
- draw modes, used to perform boolean operations on the source and destination pixels

The basic functions available in each class and the additional properties are described below.

Text

The PC/GEOS text primitives include **GrDrawChar** and **GrDrawText**. There are many properties available for text primitives:

- font, such as URW Roman and URW Sans
- point size, up to 792 point
- style, such as bold, italic, underline, etc.
- track kerning (inter-character spacing)
- space padding (width of the space character)
- vertical justification (baseline, bottom, top)

Line

The PC/GEOS line primitives include basic primitives such as **GrDrawLine**, **GrDrawPolyLine**, and **GrDrawRectangle**, as well as more complex stroked objects such as arcs, ellipses, and bezier curves (splines).

There are a variety of line properties that can be changed by the application to affect how the line is rendered. The properties particular to lines include:

- thickness, up to 1000 points
- style (solid, dashed, dotted, custom)
- end type (round, square, butted)
- join type (round, beveled, mitered)

Area

The PC/GEOS area primitives include **GrFillRect**, **GrFillEllipse**, and **GrFillPolygon**. There are a few additional properties available for the area primitives:

- border width, up to 1000 points
- border join (similar to line join type)
- polygon fill rule (odd-even or winding rules)

Raster

There are two basic raster primitives available under PC/GEOS: **GrDrawBitmap** and **GrBitBlt**. Every bitmap structure begins with a header. This header contains such parameters as compaction methods, color palettes, width, height, and resolution. The data for the bitmap follows. The bitmap can then be drawn on and moved about the screen.

Color

PC/GEOS supports the use of color, either in monochrome, 4-bit, 8-bit, or 24-bit mode. Colors are specified using indexes or RGB values. Any particular object can be assigned a palette of colors appropriate for the number of color bits being used.

When colored objects are displayed on a monochrome video display, colored pixels are mapped to black and white pixels in either solid or dithered mode. If the application chooses a solid mapping, then any color other than white is translated to black. If the

application chooses dithering, then PC/GEOS calculates the luminosity of the given color and produces an appropriate shading using black and white pixels.

Graphics Strings

The PC/GEOS graphics system has the capability to "retain" graphics operations instead of executing them. That is, it can store an opcode and the arguments that represent a call to a function in the graphics system to memory, to a file, or to any stream device. This is used by the system to implement a part of the clipboard, and as a format for print jobs sent to the spooler.

The use of these graphics strings is largely transparent to the application creating them. The application makes a call to redirect graphics commands to a string. After this call, every graphic function is converted into an opcode and argument pair and stored in a supplied buffer. One additional call completes the graphics string and stops the redirection. There are additional graphics functions available to play back these graphics strings. Of course, this redirection process has no impact on other applications or user interface activities. An example of the use of graphics strings is briefly described below, in the printing section.

Font Technology

The appearance of printed text is very important, since high quality output is one of the top priorities in the design of the system. Because there are many competing technologies in the area of font rendering, PC/GEOS implements a flexible strategy designed to accommodate any font technology.

PC/GEOS achieves this flexibility with font drivers. PC/GEOS includes a driver for the Nimbus-Q outline font engine, developed by The Company. This engine renders fonts stored in the URW digital font format. The URW library includes 1600 different typefaces. There are plans to include drivers for other families of outline-defined typefaces (such as Bitstream) in the future.

There are two basic types of font technologies used in graphical environments: bitmap fonts and outline fonts. Bitmap fonts (as used on the Macintosh and Windows) have the advantage of fast drawing speed, but suffer greatly in the quality of type, especially when the type is scaled. Outline fonts (as used in Postscript) can be of extremely high quality, but are traditionally much slower to render. PC/GEOS uses a combination of these technologies to achieve both fast drawing speed and high output quality. The actual text drawing code in the video drivers require characters to be formatted as bitmap data, for fast drawing speed. If bitmap data is not available for a given character/scale factor/rotation angle combination, a font driver is called to build the character.

PC/GEOS operates with any combination of outline and bitmap fonts.

Video Drivers

The PC/GEOS graphics system uses video drivers to provide a layer of device independence, separating the device-dependent portion of the graphics code from the common portion. There are a number of drivers currently available which cover the most popular video cards: MCGA, CGA, Hercules, EGA, VGA, and Everex Viewpoint™. Other drivers will be available soon for the 8514 and 34010 boards, as well as full-page displays.

The system is capable of supporting and making use of intelligent video devices. The system detects when a device is capable of rendering more complex objects (like ellipses) and passes the command to the driver at a higher level.

Printing

PC/GEOS maintains a single imaging model for both screen output and printed output. This ensures that what users see on screen is an accurate representation of what is printed on their hardcopy devices.

How an Application Prints

The user interface library includes a generic **SpoolPrintControl** object which provides the user interface aspects of printing. This object displays the options available for printing (such as paper size, which printer to use, etc.) and also implements all the necessary communication with the system print spooler.

When the user has selected the desired print options, and starts the printing process, the **SpoolPrintControl** object creates a spool file and sets the graphics system into "retain" mode. (See "Graphics Strings" above) The application is then sent a message to draw its document. Any graphics function can be used by the application to render the document, and all the function calls are saved into the spool file. When the application has completed rendering all pages of the document, the **SpoolPrintControl** object closes the spool file and signals the print spooler to add the job to its queue. It is important to note that an application may, but is not required to, use the same code to draw its document, whether for screen display or printing. This substantially simplifies WYSIWYG application development.

Print Spooler

The system print spooler has two main functions: to maintain the print queue and to build the necessary information to send to the printer driver.

When an application finishes rendering its document, the **SpoolPrintControl** object passes the spool file to the spooler. This new print job is added to the print queue for the device the user has chosen.

The spooler maintains one queue for each device. If no queue is present for the device the user has specified (when the first job is printed, for example), the spooler creates a queue structure and starts an additional thread of execution to deal with printing jobs in that queue. This means that the spooler is capable of handling multiple printing devices connected to the system, and uses the multitasking capabilities of PC/GEOS to support printing to multiple devices simultaneously.

When a print job has reached the beginning of the queue, the spooler begins processing the print job. If the printer is intelligent, the spooler passes the spool file containing the graphics string to the print driver for processing. For most "dumb" raster devices, such as dot matrix printers, the spooler allocates a bitmap which represents a portion of the page, and draws the graphics string for the first page into the bitmap. After that portion of the page is built, it is sent on to the printer driver, which sends it to the printer. This is repeated until the entire page has been built and sent.

If the user has specified that the document is in landscape format, the spooler automatically rotates the document to best fit on the paper loaded in the printer. Likewise, if there is a mismatch between the document size specified in the application and the paper size loaded into the printer, the spooler "tiles" the document onto separate sheets of paper so that the entire document is printed. None of these functions require any work from the application. This means that functions such as sideways printing fall out as a function of the spooler — no specific action is required of the application.

Printer Drivers

The printer drivers support a wide variety of hardcopy devices, from dumb raster printers to intelligent printers incorporating a page description language such as Postscript.

Each printer driver is written to support a number of different print modes. These include graphics and text modes, each available in high, medium, and low print qualities. The text modes are provided to take advantage of the ASCII text printing modes that are provided on most printers, as a fast alternative to graphics printing. These text modes use only those fonts that are available on the device, and ignore any requests for graphics output.

The graphics modes print both text and graphics, using the outline fonts provided with the system to render the characters to the resolution of the printing device. Most printer drivers support a high and a low resolution to give the user the option of a quick draft or a slower high-quality printout.

Print drivers are provided for most dot-matrix models, PostScript compatible printers, and HP LaserJet compatible printers. Printer driver development continues in order to support additional printers, new printer models, and other hardcopy devices such as plotters and film recorders.

Memory Management

All applications and PC/GEOS system functions require memory. Unfortunately, there is a very limited supply of memory. Due to these tight memory constraints, the memory manager makes all memory allocation and usage decisions. PC/GEOS manages memory with the following goals in mind:

- High performance, optimized for several applications running on a 640K machine with limited discarding and swapping
- Isolation of the application from the hardware memory configuration
- Dynamic use of memory

PC/GEOS memory management is based on a basic 640 Kbyte RAM model. Of this 640K, 30K to 160K are occupied by MS-DOS, depending on the version being used. A fixed 55K block in RAM contains the PC/GEOS kernel. The rest of the 640K RAM is used for the global heap, described below.

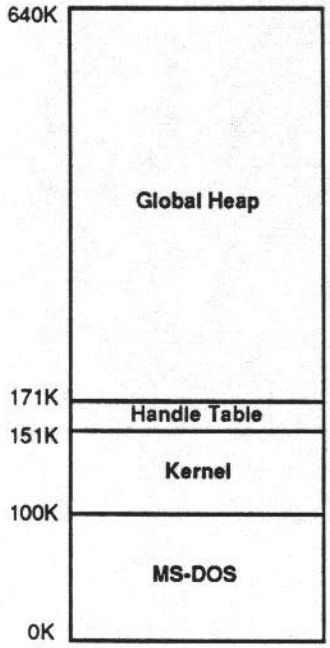


Figure 6. Typical 640 Kbyte Configuration

For those systems with more than 640 Kbytes RAM, PC/GEOS takes advantage of extended memory by treating it as a swap space. In essence, portions of memory above 640K are used as a very fast RAM disk. Expanded memory is treated similarly, providing swap space as well as an additional 48 Kbytes of heap space.

Global Heap

Memory available to applications is organized in a data structure called the "global heap." The heap varies in size from machine to machine and can change from execution to execution but never changes during a single execution of PC/GEOS. Usually, the global heap occupies approximately 450 Kbytes of memory, depending on the version of MS-DOS being used.

Because of hardware constraints imposed by some system configurations, PC/GEOS implements a segmented memory management scheme. The logical address space is viewed as a collection of individual segments, each a contiguous range of memory within the global heap. Each segment has a name and a length, so memory is addressed by both the segment name (called a "handle") and an offset within the segment.

The memory management routines in PC/GEOS are extensible to handle linear address spaces without requiring modification of existing applications.

Blocks and Handles

Memory is allocated in segments, called blocks. Each block may contain up to 64 Kbytes of code or data. The internal composition of these blocks is described in "Using Blocks" below. Each block is referenced using a "handle" which uniquely identifies that block. The system manages the blocks via a handle table located in memory between the kernel and the global heap. The number of handles contained in the handle table is determined by system configuration and is user-definable. Specifically, a handle is a 16-bit value representing the address of a block's handle table entry.

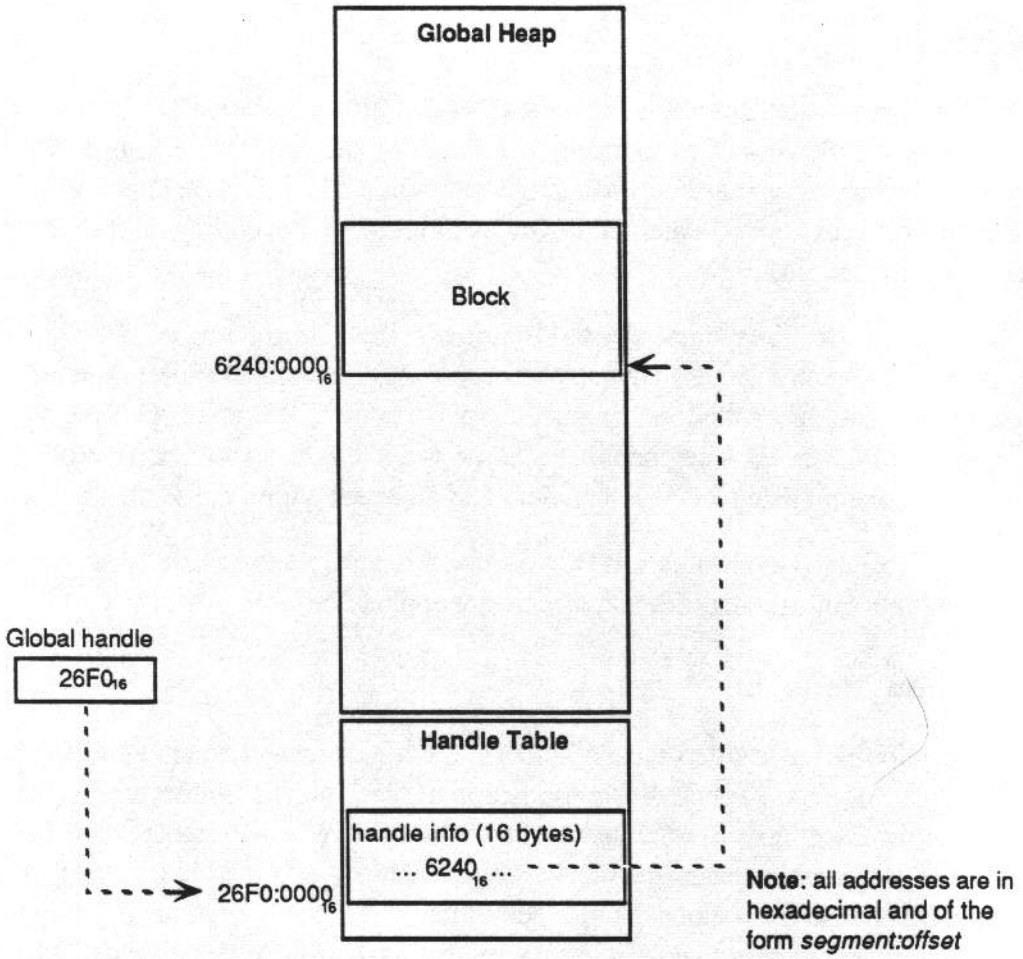


Figure 7. The Global Heap

Blocks have a number of attributes. Some of the more important attributes include whether a block should be fixed in memory, discardable, and/or swappable.

A block can be fixed or moveable. If it is fixed, the block stays exactly where it is in memory until specified otherwise. If it is moveable, PC/GEOS may move it around within the global heap when it is not actively being used.

A block, typically containing code, can be designated as being discardable. These read-only blocks are flushed from memory whenever PC/GEOS requires more free space. A block is discarded by freeing the memory associated with the block and setting the appropriate flag in the handle table. Further attempts to use the block cause it to be read back into newly allocated memory.

Blocks can also be designated as swappable. A block is swapped by copying the block to extended memory or to disk and then freeing the memory associated with the block. Subsequent attempts to use the block results in the block being read back into newly allocated memory. Swapping is logically transparent to the user.

Allocation of Blocks

Movable blocks are allocated from the top of the heap using a first-fit method. If there is not enough contiguous free memory on the global heap, PC/GEOS attempts to shuffle the contents of the heap in order to place all free memory together in one large block. If this does not liberate enough space, blocks are discarded or swapped, and the heap is again compacted to produce a free block of sufficient size. Due the multitasking nature of PC/GEOS, this compaction process occurs in parallel with other tasks.

There are many routines provided to accommodate all aspects of memory allocation. Listed below are the most frequently used:

MemAlloc: This routine allocates a given number of bytes on the global heap. Pass it the number of bytes desired and block attributes (fixed/movable, etc.). It returns a handle to the newly allocated block, the address of the block allocated if it is fixed, and an error flag if there is not enough free space.

MemReAlloc: This routine changes the size of a block on the heap or reallocates space for a block which has been discarded. Pass it the handle of the block to reallocate space for, the new size desired (or 0 to reallocate space for a discarded block), and some parameters. It returns the handle of the reallocated block and an error flag.

MemFree: This routine releases the memory allocated for a block and frees the handle associated with it. Pass it the handle of the block you wish to free.

Because blocks are subject to movement any time a new block or more memory is requested, applications should not store segment values (addresses) for its blocks. The handle table is updated automatically, so if applications reference blocks of memory solely by their handles the program need not worry about the activity of the global heap.

Accessing Memory

Accessing memory is straightforward. Given the handle of the block you would like to access, merely call a kernel routine to lock the block. PC/GEOS prevents the block from being moved in the global heap and returns an absolute address for you to use. Once you have completed operations involving that particular block, you may either unlock the block and leave it in memory for PC/GEOS to move as needed, or you may immediately discard the block. Described below are the most frequently used routines:

MemLock: This routine returns the absolute address (a 16-bit segment value) of a block on the heap and locks the block, preventing it from being moved. Pass the handle of the block to lock.

MemUnlock: This routine unlocks the given block, allowing it to be moved, discarded, or swapped. Pass the handle of the block to unlock.

Using Blocks

An application may use a block in any manner it chooses. However, using blocks for small pieces of data or temporary buffers is inefficient and wastes a global handle. Therefore, an application may create a heap within a block, called a local memory block. By avoiding some of the main heap routines, the application saves space and global handles and can keep all of its buffers in one block for easier access. The Local Memory Manager provides routines to access and dispose of these local memory blocks. The Local Memory Manager is used extensively in the management of UI objects.

To create a local memory heap the application first allocates and locks a block of memory. Then it calls **LMemInitHeap** to initialize a heap at a certain offset into the block. The heap occupies the entire block from the given offset to the end of the block. The **LMemBlockHeader** structure is placed at the beginning of the block.

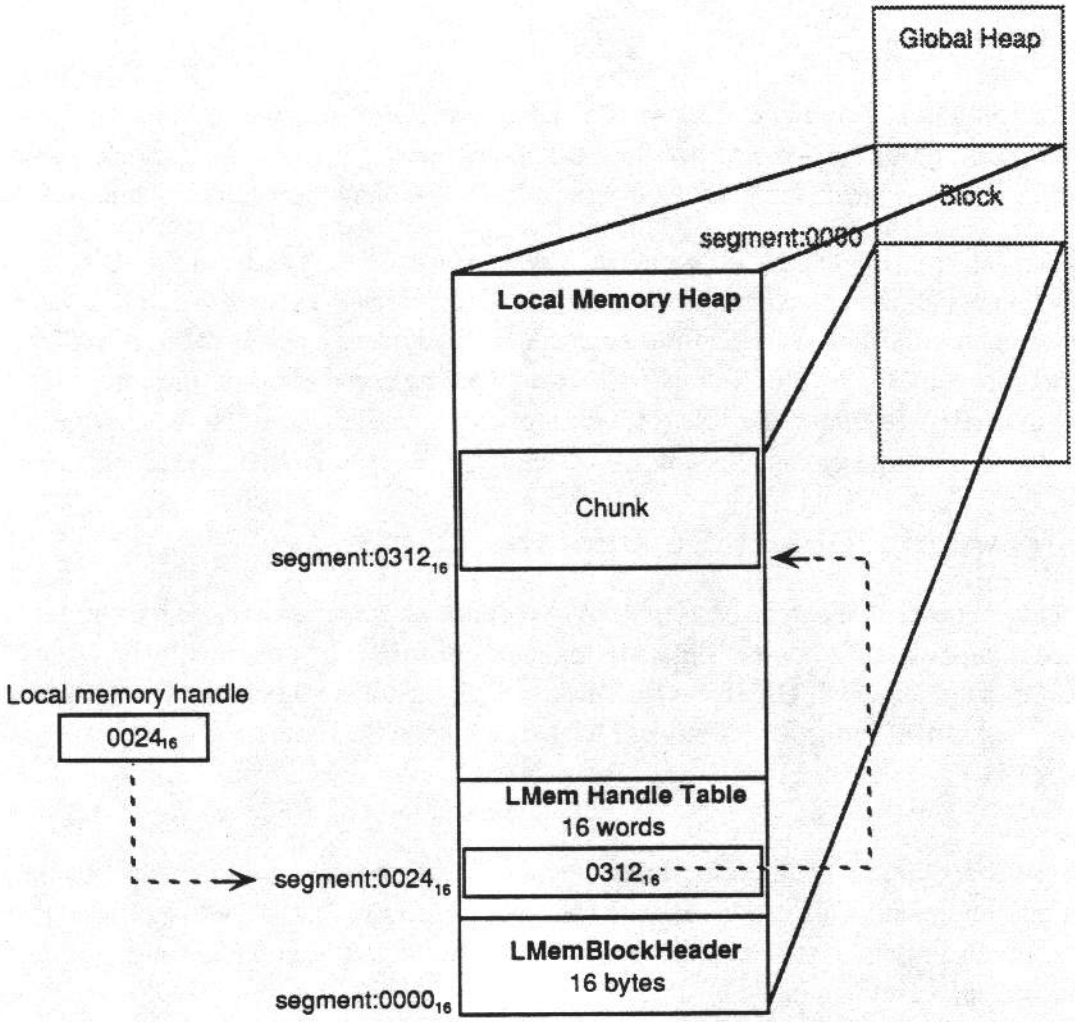


Figure 8. A Local Memory Block

The application can then create as many "chunks" of memory on the heap as it wishes, resizing them, deleting them, and otherwise manipulating them. To create a chunk, it calls **LMemAlloc**, passing the length of the desired chunk. The routine allocates space on the local heap and returns a handle for the chunk. Such a handle is called a local memory handle or chunk handle, to distinguish it from a global memory handle.

The chunk handle is used as an offset into the block to the location where the chunk's true offset value is stored. This means that, given a chunk's handle, only one instruction is required to determine its offset. This offset value is valid only between calls to LMem routines. It is not safe to assume that a chunk occupies the same location in the local heap after any LMem call. Therefore, store the handle, not the offset value.

An application may resize a chunk by calling **LMemReAlloc**. To free a chunk on the local memory heap, it calls **LMemFree**. When the application no longer needs the local heap, it may call **MemFree** to free the entire block.

Object Blocks

An object block is simply a block on the global heap which contains objects, such as user interface objects. To organize the storage of these objects within the block, a local memory heap is created, and each object is allocated a chunk on the local memory heap.

Therefore, to reference any object in the system, you need the handle of the object block in which the object resides and the chunk handle for the chunk which holds the object's instance data. Such a global handle and chunk handle pair is called an object descriptor, or OD. Within PC/GEOS, there are naming conventions to indicate that two words constitute an object's OD. **^lcx:dx**, for example, indicates that the **cx** register holds the 16-bit global handle, and the **dx** register holds the 16-bit chunk handle.

Dereferencing Object Descriptors

An object descriptor can be dereferenced to produce an actual address which can be used to manipulate an object's instance data. For example, assume that **^lbx:si** is the OD of the object. Call **ObjLockObjBlock** to lock the object block on the global heap. **ObjLockObjBlock** is a derivative of **MemLock** specifically used with object blocks.

```
call    ObjLockObjBlock   ;returns ax = segment of block
mov     ds, ax            ;set *ds:si = instance data of object
```

***ds:si** is another convention for describing an OD. Above it is used to show that the **ds** register contains the actual segment (physical address) of the object block, and that the **si** register contains the chunk handle. Next, dereference the chunk handle, to produce an offset within the block:

```
mov     di, ds:[si]       ;set ds:di = instance data for object
```

Another convention, **ds:di**, indicates that the **ds** register contains a segment value, and the **di** register contains the offset within the block at which the chunk is located. This register pair can then be used by any of the processor's instructions which manipulate memory, such as **and**, **or**, and **mov**.

Please do not infer from this example that it is acceptable for one object to directly modify another object's instance data. This would violate the principles of object-oriented programming. Whether you are writing code for an application's ProcessClass object or for a UI object's method handler, PC/GEOS defines which dereferencing operations are permitted.

Threads

PC/GEOS uses the term "thread" to denote a thread of execution. A thread is an executable unit; it has an entry in the handle table and is identified by this handle. Each thread has its own program counter, set of registers, and stack — essentially the state of the CPU the last time it was executing that particular thread.

Since PC/GEOS is a multitasking operating system, there can be several threads in existence at a time, though only one thread can be executing at once. By managing these threads and switching rapidly between them, PC/GEOS can perform several tasks at once.

PC/GEOS manages threads in order to

- have fast context switches between threads
- isolate an application from the multiple application environment
- feature pre-emptive multitasking
- schedule to maximize user response while maximizing resource utilization

There are three primary threads which are always running during a session with PC/GEOS: the Input Manager, the user interface, and an application thread.

Thread Priorities

Since PC/GEOS is a multitasking operating system, it implements a methodology for determining which thread to execute and for how long. All executable threads are held in the run queue, a list containing the handles of threads which are awaiting CPU time.

A thread has two priorities: its base and current priority. The base priority is defined at creation, while the current priority starts at the base priority and degrades with recent CPU usage. Normally, PC/GEOS assigns default base priorities to application threads. However, if you create your own threads for a specific purpose and would like a higher or lower priority than given, the base priority can be adjusted accordingly.

As a thread receives CPU time, it is interrupted ten times per second. Its current priority is decreased, and if it is still the thread with the highest current priority, it is allowed to continue. In this manner, every thread is allowed its share of CPU time. Because PC/GEOS features true pre-emptive multitasking, a new process with higher priority than the current thread may preempt its execution at *any* time.

The input manager thread is the most important because PC/GEOS is designed with user interaction as its foremost concern. The input manager thread is given more

preference all other threads, so that the user never experiences "lag time" between a mouse movement or a button press and the expected response.

Interaction Between Threads

Objects send messages to each other. If the objects are run by the same thread, the message is handled immediately. However, the sender and receiver are often executed by the different threads. For example, an application's ProcessClass object (run by the application thread) might send a message to a UI object (run by the UI thread) instructing it to disable itself. There are two ways the application may invoke the disable method: by sending a message to the UI thread's queue and by blocking on the method call.

Most threads have a queue. Messages sent to an object are placed on the queue for the thread running that particular object. When the thread receives processor time, it selects the first method on the queue and invokes it. Sending a message to the object's queue allows the application to continue executing code. The object receiving the message handles it independently of the application thread.

If it is vital that the targeted object deal with the message before any other action is taken, the application may block on the method call. In other words, the application sends the object the appropriate message and waits until the object is done before continuing execution.

Sharing Resources

In a multitasking environment, disaster lurks if there is not a way to manage shared resources, to make sure that no two threads attempt to access and modify the same file, for instance. In most cases, the kernel handles the synchronization of resources. However, libraries, drivers, and applications with multiple threads may need to be more careful.

For example, an application may block on a semaphore. A semaphore is essentially a flag which alerts threads if a resource is being used. For example, suppose the application is attempting to write to a file. With one command, **P(file)**, the application thread may attempt to secure that file. If the file is being used, the thread is "blocked," waiting on the semaphore "**file.**" When the **file** semaphore is released, the thread is "woken up" and given exclusive access to the file. After the application has completed its work, it releases the **file** semaphore with **V(file)**.

System Services

Virtual Memory Management

All parts of PC/GEOS use the heap to manage memory. For permanent storage, however, data must be moved into a disk file. Disk files are also used when it is impractical to load an entire file into memory. The heap is useful because it provides a higher level interface to memory — it partitions memory into smaller pieces which can be resized and moved transparently. Disk files, on the other hand, are long streams of data. While this is useful in some contexts, most applications find disk files cumbersome. The virtual memory (VM) interface, a set of routines in the kernel, provides a higher level view of disk files and automatically caches parts of a disk file in memory.

The Virtual Memory Manager

- allows applications to refer to blocks of storage via VM block handles, which uniquely identify a block of data, whether it is on disk or in memory
- automatically reads in VM blocks when needed, caches them in memory, and writes dirty (modified) blocks to disk
- manages disk files such that information can be read and written quickly
- manages VM files as shared resources so that multiple processes can access a VM file simultaneously

The Virtual Memory Manager is essentially a disk-based heap manager. VM blocks can be allocated and utilized for storage. These blocks are accessed in the same manner as regular memory blocks, via VM block handles. When changes have been made to VM blocks, the Virtual Memory Manager must be informed so that it may update these "dirty" blocks when permanent storage is desired.

VM files may contain object blocks and are very useful for storing documents. For example, GeoDraw and GeoWrite place documents in VM files.

Database Library

Many applications manage data-structures which need to be saved to disk as a data file. Management of these data structures is a crucial issue in maintaining good system performance. Furthermore, in many cases the data structures required for a data file are larger than can be reasonably held in memory.

The PC/GEOS Database Manager, a dynamically loadable library, provides a clean interface for an application to manage any type of data file while still allowing PC/GEOS to reclaim memory when needed. Updating a data file is fast and efficient. The Database Manager loads into memory only those portions of the data file which are needed by the application, marks as dirty all portions which have changed, and allows all unchanged data to be discarded. The kernel can flush data to disk when it does garbage collection. The database library essentially combines the Local Memory Manager and the Virtual Memory Manager.

Items and Groups

Since the Database Manager deals with data structures, it accommodates the logical organization of such files. The data structure is considered an item, and a collection of items comprises a group. The database manager implements each item as a chunk and each group as one or more blocks in memory, so the application should designate groups with an upper bound in mind.

Document Control Object

The concept of a Document Control Object follows naturally from the previous two sections. Since the operating system handles standard disk files as well as structured disk files, it can also liberate applications from the remaining troublesome aspects of file maintenance: the file menu, dialog boxes, and saving or discarding changes.

The Document Control Object creates and manages the standard file menu and related dialog boxes. Among these options are Save, Revert, Save As, and Close.

Save: All changes are saved in the document which was opened. If the document is newly created, then the user is prompted to enter a filename.

Revert: All changes to the document *since the last save* are nullified.

Save As: The user is prompted for a new filename, and all changes to the document are saved in the new file. The old document retains the original version (the equivalent of the Revert option).

Close: The document is closed, retaining both the original version (from the last save) and all changes since the last save. Thus, if the user were to open the document again, choosing Save, Save As, or Revert would produce the same results as if the user had never closed and re-opened the document.

Sound

Producing sounds in a multitasking environment is not always a straightforward task. PC/GEOS uses a sound queue so that applications need only call a routine to generate sounds. PC/GEOS provides several such routines.

The user interface emits a sound automatically when an error box appears or when the user enters illegal input. The user may disable sound by choosing the appropriate setting in the Preferences desk tool. Sound routines check this setting when executed, so applications may simply call the routines.

To generate a sound, call UserStandardSound, passing it a StandardSoundTypes type. There are six types of sounds, defined in the enumerated type StandardSoundTypes:

- SST_ERROR, beep produced when an error box appears
- SST_WARNING, general warning beep
- SST_NOTIFY, general notification beep
- SST_NO_INPUT, beep produced when a user provides illegal input
- SST_CUSTOM_BUFFER, a custom note buffer defined by the application (according to the format described in sound.def)
- SST_CUSTOM_NOTE, a custom note given a frequency and duration

Localization

Since PC/GEOS was designed with a global perspective in mind, it provides many features to make localization of applications simple. As described before in "Geometry Manager," changes in textual monikers of UI objects are handled automatically. Additional routines include:

- string comparisons, accounting for accented characters and special cases
- string manipulation, such as conversion to/from upper and lower case
- time formats
- date formats
- currency formats
- measurement types (english vs. metric)

Alternate keyboards will be supported through separate keyboard drivers.

Appendix A: Class Hierarchy

This is the PC/GEOS class hierarchy. Indentation indicates the relative class levels; for example, VisTextClass is a subclass of VisClass. MetaClass is the highest level in the system. Every PC/GEOS application must define a subclass of UI_Class; CalendarClass and RolodexClass are given as examples. Additionally, an application can define subclasses of any of the system-provided classes such as MetaClass or GenTriggerClass.

```
MetaClass
    ProcessClass
        UI_Class
            CalendarClass
            RolodexClass
    VisClass
        VisTextClass
        VisCompClass
            VisContentClass
              VisIsoContentClass
                  GrObjIsoContentClass
        GraphicClass
            ObjectClass
            GraphicBodyClass
    GenClass
        GenTriggerClass
          GenDataTriggerClass
        GenActiveListClass
          GenApplicationClass
          GenDisplayClass
              GenPrimaryClass
          GenDisplayControlClass
        GenFieldClass
        GenInteractionClass
          GenSummonsClass
          ImporterClass
          SpoolPrintControlClass
        GenGlyphDisplayClass
        GenTextDisplayClass
          GenTextEditClass
        GenViewClass
        GenContentClass
        GenListClass
        GenListEntryClass
        GenGadgetClass
        GenSpinGadgetClass
        GenRangeClass
        GenUIDocumentControlClass
        GenAppDocumentControlClass
        GenPrintControlClass
        GenFileSelectorClass
        GenDocumentClass
```

Appendix B: Routine Names

Listed below are all the routines available to Applications, grouped by subject and major subject.

User Interface Routines

TokenInitTokenDB
TokenExitTokenDB
TokenDefineToken
TokenGetTokenInfo
TokenLookupMoniker
TokenLoadMoniker
TokenRemoveToken
TokenGetTokenStats
TokenLoadToken
TokenLockTokenMoniker
TokenUnlockTokenMoniker
DefineTokenLow
UserCallFlow
UserLoadApplication
UserCheckAcceleratorChar
UserGetSystemStyleRun
UserCheckInsertableCtrlChar
UserGetApplicationOD
UserCallApplication
CheckForDamagedES
ECCheckUILMemOD
ECCheckODCXDX
ECCheckLMemODCXDX
ECCheckUILMemODCXDX
UserScreenRegister
UserStandardSound
UserAllocObjBlock
DrawRegionAtOrigin
UserCopyChunkOut
UserHaveProcessCopyChunkIn
UserHaveProcessCopyChunkOut
UserHaveProcessCopyChunkOver
UserFontCreateList
UserCreateListEntry
UserAddEntryToList
UserCheckProtocol
UserSetProtocol
CheckKbdShortcutList
FlowForceGrab
FlowReleaseGrab
FlowRequestGrab
FlowGainedExcl
FlowLostExcl
FlowGrabWithinLevel
FlowReleaseWithinLevel
FlowTranslatePassiveButton
FlowGetButtonFlags
FlowStartQuickTransfer
FlowEndQuickTransfer
FlowAbortQuickTransfer
ECFlowEnsureHandleNotReferenced
ECFlowEnsureODNotReferenced
ECFlowEnsureWinNotReferenced
FlowGrabMouse
FlowForceGrabMouse
FlowReleaseMouse
FlowGrabKbd
FlowForceGrabKbd
FlowReleaseKbd
FlowAddButtonPrePassive
FlowRemoveButtonPrePassive
FlowAddButtonPostPassive
FlowRemoveButtonPostPassive
FlowAddKbdPrePassive
FlowRemoveKbdPrePassive
FlowAddKbdPostPassive
FlowRemoveKbdPostPassive
FlowAddToGeneralNotificationList
FlowRemoveFromGeneralNotification List
FlowIgnorePtrEvents
FlowSendEnterLeavePtrEvents
FlowSendAllPtrEvents
FlowTranslatePassiveButton
FlowGetButtonFlags
FlowBumpMouse
FlowSetPtrImage
FlowCheckKbdShortcut
FlowGetTarget
FlowTestIfActiveOrImpliedWin
FlowHoldUpInput
FlowResumeInput
FlowDisableHoldUpInput
FlowEnableHoldUpInput
FlowSendFileChange
VisGetSize
VisGetBounds
VisGetCenter
VisCallParent
VisFindParent

VisSwapLockParent

UI Routines (Cont'd)
VisCallChildren
VisCallChildUnderPoint
VisCallFirstChild
VisCallNextSibling
VisTestPointInBounds
VisQueryGWin
VisQueryParentGWin
VisGetParentGeometry
VisHandleDesiredResize
VisHandleMinResize
VisHandleMaxResize
VisMarkInvalid
VisMarkInvalidOnParent
VisResize
VisMove
VisFindMoniker
VisGetMonikerSize
VisDrawMoniker
VisInsertChild
VisConvertSpecVisSize
VisConvertCoordsToRatio
VisConvertRatioToCoords
VisGetMonikerPos
VisConvertSpecVisSize
VisTestMoniker
VisUpdateSearchSpec
VisCheckMnemonic
VisBuildSetEnabledState
VisSendVisBuild
VisSendUpdateVisBuild
VisGetVisParent
VisAddChildRelativeToGen
VisGetSpecificVisObject
VisMarkFullyInvalid
VisCheckIfVisBuilt
VisCheckIfVisGrown
VisCheckIfFullyEnabled
VisTakeGadgetExclAndGrab
VisGrabMouse
VisForceGrabMouse
VisReleaseMouse
VisGrabKbd
VisForceGrabKbd
VisReleaseKbd
VisAddButtonPrePassive
VisRemoveButtonPrePassive
VisAddButtonPostPassive
VisRemoveButtonPostPassive
VisIgnorePtrEvents
VisSendAllPtrEvents
VisSendEnterLeavePtrEvents
VisGrabFocusExcl
VisReleaseFocusExcl
VisGrabTargetExcl
VisReleaseTargetExcl
VisReleaseAll
VisCheckIfUsable
VisGetGenBranchInfo
IfFlagSetCallVisChildren
IfFlagSetCallGenChildren
VisSendCalcNewSize
VisSendMove
VisRemove
VisSetNotRealized
VisNavigateCommon
VisNavigateTestForIDCommon
VisVisBuild
VisCompGetCenter
VisCompCalcNewSize
VisCheckOptFlags
CheckVisAssumption
CheckVisMoniker
ECVisStartNavigation
ECVisEndNavigation
ECCheckVisFlags
ECCheckVisCoords
VisInitialize
VisCompInitialize
UserTextMapDefaultStyle
UserTextFindDefaultStyle
UserTextMapDefaultRuler
UserTextFindDefaultRuler
GenGetDisplayScheme
GenFindMoniker
GenDrawMoniker
GenGetMonikerSize
GenScanHints
GenScanBothHintLists
GenCallSystem
GenCallParent
GenGetApplicationOD
GenCallApplication
GenCallProcess
GenCallApplicationViaProcess
GenFindParent
GenSwapLockParent
GenCallChildren
GenCallNextSibling
GenInsertChild
GenCopyChunk
GenProcessGenAttrsBeforeAction
GenProcessGenAttrsAfterAction
GenMarkDirty
GenAddChildUpwardLinkOnly

GenSetUpwardLink

UI Routines (Cont'd)

GenRemoveDownwardLink
GenFindObjectInTree
GenCheckKbdAccelerator
GenFindTopModalWindow
GenGetByte
GenSetByte
GenGetWord
GenSetWord
GenGetTwoWords
GenSetTwoWords
GenGetDWord
GenSetDWord
GenUnbuild
GenCheckIfFullyUsable
GenCheckIfFullyEnabled
GenCheckIfSpecGrown
CheckGenAssumption
EnsureNotUsable
ECEnsureInGenTree
GenSetupNormalizeArgs
GenReturnNormalizeArgs
UserRegisterTransfer
UserUnregisterTransfer
UserQueryTransfer
UserRequestTransfer
UserDoneWithTransfer
UserGetNormalTransferInfo
UserGetQuickTransferInfo
UserGetUndoTransferInfo
UserGetUITransferFile
UserAddTransferNotify
UserRemoveTransferNotify
UserDoDialog
UserStandardDialog
WinGetWinBounds
WinGetWinScreenBounds
WinGetMaskBounds
WinOpen
WinClose
WinChangePriority
WinScroll
WinBitBlt
WinStartUpdate
WinEndUpdate
WinAckUpdate
WinInvalReg
WinInvalRect
WinLocatePoint
WinGetInfo
WinSetInfo
WinApplyRotation
WinApplyScale
WinTransformCoord
WinUnTransformCoord
WinSetTransform
WinApplyTransform
WinSetNullTransform
WinGetTransform
WinMouseGrab
WinMouseRelease
WinMovePtr
WinGrabChange
WinUngrabChange
WinChangeAck
WinEnsureChangeNotification
WinMove
WinResize
WinDeathAck
WinBranchExclude
WinBranchInclude
WinApplyTranslation
WinForEach
WinSuspendUpdate
WinUnSuspendUpdate
WinIntTransCoord
GrSetPtr
GrHidePtr
GrShowPtr
GrMovePtr
GrIntDrawLineLow
GrIntFillRectLow
GrIntDrawRectLow
GrIntTransCoord
GrIntStoreBytes
GrIntCreateBMHelp
GrIntGStateTransCoord

Graphics Routines

GrCopyDrawMask
GrMapColorToGrey
GrGetDefFontID
GrCharMetrics
GrFontMetrics
GrCharWidth
GrTextWidth
GrTextWidthWBFixed
GrTextPos
GrGetTextWrapInfo
GrTextObjCalc
KLAddWidth
KLUpdateFieldHeightVars
KLCallStyleCallBack
GrGetBitmap

GrCallFontDriver

Graphics Routines (Cont'd)

GrTransForFont
GrSetClipRect
GrCreateState
GrDestroyState
GrStartExclusive
GrEndExclusive
GrTransCoordWWFixed
GrUnTransCoordWWFixed
GrMulWWFixed
GrMulWWFixedPtr
GrSDivWWFixed
GrUDivWWFixed
GrSqrWWFixed
GrSqrRootWWFixed
GrQuickSine
GrQuickCosine
GrQuickArcSine
GrBitBlt
GrSetStringPos
GrDrawRegion
GrDrawRegionAtCP
GrTransformCoord
GrUnTransformCoord
GrPlayString
GrPlayStringAtCP
GrMapColorIndex
GrMapColorRGB
GrGetClosestRGB
GrGetPaletteMap
GrLockPalette
GrUnlockPalette
GrGetPalette
GrCreateBitmap
GrDestroyBitmap
GrSetPrivateData
GrCreateRectRegion
GrCreatePolygonRegion
GrGetDrawMode
GrGetLineColor
GrGetAreaColor
GrGetTextColor
GrGetLineMask
GrGetAreaMask
GrGetTextMask
GrGetLineColorMap
GrGetAreaColorMap
GrGetTextColorMap
GrGetTextSpacePad
GrGetTextStyle
GrGetTextMode
GrGetLineWidth
GrGetLineEnd
GrGetLineJoin
GrGetLineStyle
GrGetMiterLimit
GrGetCurPos
GrGetInfo
GrTextObjCalc
GrDestroyString
GrLoadString
GrBeginString
GrGetElement
GrDrawString
GrDrawStringAtCP
GrTextPosition
GrGetTransform
GrSetBitmapRes
GrGetBitmapRes
GrClearBitmap
GrGetBorderWidth
GrGetBorderJoin
GrGetFont
GrIsPointInPolygon
GrGetBitmapSize
GrChunkRegOp
GrPtrRegOp
GrMoveReg
GrGetPtrRegBounds
GrTestPointInReg
GrTestRectInReg
GrBuildRegion
GrEnumFonts
GrIsFontAvail
GrFindNearestPointsize
GrObjMarkFileEntry
GrObjUnMarkFileEntry
BMCallBack

Memory Mgmt. Routines

MemEMMSetup
MemAlloc
MemReAlloc
MemFree
MemInfo
MemModify
MemLock
MemUnlock
MemInfoHeap
HandleP
HandleV
MemPLock
MemUnlockV
MemThreadGrab

MemThreadGrabNB

Mem Mgmt Routines (Cont'd)

MemThreadRelease
MemVerifyHeap
MemDerefDS
MemDerefES
MemOwner
MemAllocSetOwner
LMemInitHeap
LMemAlloc
LMemAllocHere
FarLockInfoBlock
FarUnlockInfoBlock
MemIntAllocHandle
LMemFree
LMemReAlloc
LMemInsertAt
LMemDeleteAt
LMemExists
LMemCompactHeap
LMemAllocTempChunk
LMemFindTempChunk
LMemFreeTempChunk
ECCheckLMemChunk
ChunkArrayCreate
ChunkArrayElementToPtr
ChunkArrayPtrToElement
ChunkArrayAppend
ChunkArrayDelete
ChunkArrayGetCount
ChunkArrayEnum
ChunkArrayZero
VMLock
VMUnlock
VMAlloc
VMFind
VMFree
VMDirty
VMModifyUserID
VMInfo
VMGetDirtyState
VMGetMapBlock
VMSetMapBlock
VMGetMapExtra
VMSetMapExtra
VMOpen
VMUpdate
VMClose
VMGetAttributes
VMSetAttributes
VMStartExclusive
VMEndExclusive
VMSetThreadVMFile
VMGetThreadVMFile
VMGetHeader
VMSetHeader
VMSetReloc
VMAttach
VMDetach
VMMemBlockToVMBlock
VMVMBlockToMemBlock
VMSave
VMSaveAs
VMRevert
SwapInit
SwapWrite
SwapRead
SwapFree
MemAddSwapDriver
MemExtendHeap

File and Disk Managment Routines

FileCreateDir
FileDeleteDir
FilePushDir
FilePopDir
FileGetCurrentPath
FileSetCurrentPath
FileOpen
FileCreate
FileClose
FileCommit
FileCreateTempFile
FileDelete
FileRename
FileRead
FileWrite
FilePos
FileTruncate
FileSize
FileGetDate
FileSetDate
FileDuplicateHandle
FileForceDuplicateHandle
FileLockRecord
FileUnlockRecord
FileGetDiskHandle
FileEnum
FileGetAttributes
FileSetAttributes
FileGetSystemInfo
FileGetVolumeInfo
FileSetVolumeName

FileGetVolumeFreeSpace
FileSetStandardPath

File and Disk Mgmt Routines (Cont'd)

FileCreateLongName
FileOpenLongName
FileCheckValidLongName
FileExistsLongName
FileMapLongNameToDosName
FileGetDosNameFromLongName
FileLocateFileInDosPath
FileCopy
FileEnum
FileGrabSystem
FileReleaseSystem
FileInt21
DiskValidateFar
DiskUnlock
FileDriverGone
FileFindFirst
FileLocateVolumeFar
FileRecordError
FileForEach
FileForEachPath
FileGetDestinationDisk
DiskGetBootSector
DiskCopy
DiskFormat
DiskRegisterDisk
DiskRegisterDiskWithoutInformingUser
OfAssociation
DiskReRegister
DiskEnum
DiskHandleGetDrive
DiskHandleGetVolumeName
DiskVolumeNameGetDiskHandle
DiskGenerateSysId
DiskCheckDiskValid
DiskWritable?
DiskInUse?
DiskValidateFar
DiskUnlock
DriveGetStatus
DriveGetDefaultMedia
DriveTestMediaSupport
DriveReadSectors
DriveWriteSectors
DriveReturnStatusTable
DriveLock
DriveUnlock
DosExecFetchPaths

Geode (Executable File) Management Routines

GeodeLoad
GeodeUseDriver
GeodeFreeDriver
GeodeUseLibrary
GeodeFreeLibrary
GeodeForEach
GeodeFind
GeodeFindLast
GeodeInfo
GeodeGetGeodeVersion
ProcInfo
GeodeGetProcessHandle
GeodeGetAppObject
GeodeGetUIData
GeodeSetUIData
GeodeGetPermName
GeodeInfoDriver
GeodeInfoDefaultDriver
GeodeSetDefaultDriver
GeodeGetResourceHandle
GeodeGetGeodeResourceHandle
GetResourceHandleFromDS_BX
GeodeLoadResourceAt
GeodeAllocQueue
GeodeFreeQueue
GeodeDispatchFromQueue
GeodeInfoQueue
GeodeFlushQueue
ThreadExit
ThreadCreate
ThreadInfo
ThreadModify
ThreadAttachToQueue
ThreadPrivAlloc
ThreadPrivFree
ThreadHandleException
ThreadInfoQueue
GeodeLockResource
ProcCallModuleRoutine
ProcCallLibraryRoutine
ProcCallFixedOrMovable
ObjMessage
ProcBroadcastEvent

Object System Routines

ObjInitDetach
ObjIncDetach
ObjEnableDetach
ObjAckDetach

Object Sys. Routines (Cont'd)

ObjLinkCallNextSibling
ObjLinkCallParent
ObjLinkFindParent
ObjCompFindChild
ObjCompAddChild
ObjCompRemoveChild
ObjCompMoveChild
ObjCompProcessChildren
ObjCallInstanceNoLock
ObjCallInstanceNoLockES
ObjCallClassNoLock
ObjCallSuperNoLock
ObjInstantiate
ObjLockObjBlock
ObjDuplicateBlock
ObjFreeDuplicate
ObjFreeChunk
ObjIncInUseCount
ObjDecInUseCount
ObjDoRelocation
ObjDoUnRelocation
ObjResizeMaster
ObjInitializeMaster
ObjInitializePart
ObjGetFlags
ObjSetFlags
ObjAssocVMFile
ObjUnAssocVMFile
ObjIncInUseCount
ObjDecInUseCount
ObjSwapLock
ObjSwapUnlock
ObjSwapLockParent
ObjTestIfObjBlockRunByCurThread
ObjSaveBlock
ObjMapSavedToState
ObjMapStateToSaved
ObjIsObjectInClass

Miscellaneous Error-Checking Routines

FatalError
ECCheckMemHandle
ECCheckMemHandleNS
ECCheckThreadHandle
ECCheckProcessHandle
ECCheckResourceHandle
ECCheckGeodeHandle
ECCheckDriverHandle
ECCheckLibraryHandle
ECCheckGStateHandle
ECCheckWindowHandle
ECCheckQueueHandle
ECCheckLMemHandle
ECCheckLMemHandleNS
LMemValidateHeap
LMemValidateHandle
ECVMHandleVMFileOverride
ECVMCheckFileHandle
ECVMCheckStrucs
ECVMCheckBlkHandle
ECVMCheckMemHandle
ECVMCheckBlkHanOffset
ECCheckClass
ECCheckObject
ECCheckLMemObject
ECCheckOD
ECCheckLMemOD
ECCheckSegment

Database Routines

DBOpen
DBUpdate
DBSave
DBRevert
DBSaveAs
DBClose
DBLock
DBUnlock
DBDirty
DBAlloc
DBReAlloc
DBFree
DBGroupAlloc
DBGroupFree
DBSetMap
DBGetMap
DBLockMap
DBInsertAt
DBDeleteAt
DBSetThreadDBFile
DBGetThreadDBFile
DBSetThreadDBGroup
DBGetThreadDBGroup

Other routines

Input Manager

ImAddMonitor
ImRemoveMonitor
ImInfoInputProcess

ImGrabInput

Input Mgr Routines (Cont'd)
ImUngrabInput
ImSetDoubleClick
ImInfoDoubleClick
ImSetPtrMode
ImForcePtrMethod
ImSetPtrWin
ImGetPtrWin
ImPtrJump
ImStartMoveResize
ImStopMoveResize
ImConstrainMouse
ImUnconstrainMouse
ImBumpMouse
ImGetMousePos
ImGetButtonState
ImMethodButtonReceipt
ImGetButtonBacklog
ImMethodKbdCharReceipt
ImGetKbdCharBacklog
ImGetKbdCharBacklog
ImSetPtrImage

System Routines
SysNotify
SysRegisterScreen
GeodeInfoSystem
WinInvalTree
WinInvalWhole
SysShutdown
SysSetExitFlags
SysLocalInfo
DosExec
UtilHex32ToAscii
SysConfig
SysGetVersion
SysGetECLevel
SysSetECLevel
SoundPlayNote
SoundPlayBuffer
ThreadBlockOnQueue
ThreadWakeUpQueue
ThreadLockModule
ThreadUnlockModule
TimerBlockOnTimedQueue
SysEnterInterrupt
SysExitInterrupt
SysCatchInterrupt
SysResetInterrupt
SysCatchDeviceInterrupt
SysResetDeviceInterrupt
TimerGetCount
TimerGetDateAndTime
TimerSetDateAndTime
SysStatistics
SysInfo
TimerStart
TimerStop
TimerSleep

Video Driver
GrCopyDrawMask
GrMapColorToGrey
WinValClipLine
WinGenLineMask
WinMaskOutSaveUnder

System Configuration
InitFileWriteData
InitFileWriteString
InitFileWriteInteger
InitFileWriteBoolean
InitFileGetData
InitFileGetString
InitFileGetInteger
InitFileGetBoolean
InitFileGetTimeLastModified
InitFileBackup
InitFileRestore

Font/Text Driver Routines
FarTextCallDriver
FarInvalidateFont
GrLockFont
GrUnlockFont

Stream Driver
StreamNotify
StreamStrategy
StreamReadDataNotify
StreamWriteDataNotify

Appendix C: File Sizes

This appendix lists, for comparison purposes, the sizes (in Kilobytes) of many of the executables which comprise PC/GEOS. Each executable is broken into many resources which are loaded into memory only when called for. For example, GeoWrite is approximately 57 K, but only about 30 K will be in memory at any given time.

Additionally, each executable has two basic sizes: The first and most visible is the static file size of the executable containing code and relocation tables. The second is the size of the executable code alone. Both sizes are given in the tables below.

C.1 Kernel

Kernel	File Size K Bytes	Code Size K Bytes
Kernel	70.3	64.0
Kernel Library	52.6	47.4

C.2 Libraries

Library	File Size K Bytes	Code Size K Bytes
Generic UI	83.4	70.9
Motif Specific UI	124.4	107.3
Database Manager	2.8	1.9
Graphic Object Library	41.6	34.0
Import Library	7.5	6.4
Print Spool Library	32.3	29.1

C.3 Applications

Application	File Size K Bytes	Code Size K Bytes
America Online	74.4	64.3
Calculator	14.0	12.8
GeoComm	59.2	54.2
GeoDex	48.3	44.9
GeoDraw	30.5	28.3
GeoManager	92.7	85.9
GeoPlanner	55.1	49.4
GeoWrite	56.6	51.3
Preferences Manager	58.2	53.9
Solitaire	13.5	13.0
Welcome	84.8	78.7

C.4 Appliances

Appliance	File Size K Bytes	Code Size K Bytes
Address Book	52.8	50.5
Banner	18.4	17.6
Calculator	14.4	13.4
Planner	45.0	40.2
Solitaire	12.3	11.2

C.5 Drivers

Driver	File Size K Bytes	Code Size K Bytes
Video Drivers		
CGA	14.4	13.9
EGA	16.7	16.1
Hercules	14.8	14.3
VGA	16.6	16.1
Super VGA	18.1	17.5
Keyboard	3.9	3.5
DOS Driver	Avg: 3.2	Avg: 2.5
Mouse Drivers	Avg: 1.5 K	Avg: 1K
Printer Drivers	Most between 3 and 6 K	Most between 2.4 and 4 K